Technische Hochschule Nürnberg Georg Simon Ohm
Fakultät Informatik

IT-Projektbericht

OHMComm
Entwicklung eines plattformunabhängigen Frameworks zur Audioübertragung

Verfasser
Daniel Stadelmann
Jonas Ziegler
Kamal Bhatti

Datum
10. Februar 2016

Betreuer
Prof. Dr. M. Teßmann

Zusammenfassung

OHMComm ist ein plattformunabhängiges Audiokommunikationsframework, dass im Rahmen des IT-Projekts an der Technischen Hochschule Nürnberg entwickelt wurde. Das Ziel des Frameworks ist es sämtliche Funktionalität, die für eine Kommunikation benötigt wird, zur Verfügung zu stellen. Die Kommunikation erfolgt dabei als Direktverbindung über das Real-Time Transport Protokoll (RTP), welches auf UDP basiert. RTP stellt ein standardisiertes Protokoll für die Audio- und Videoübertragung dar, welches von der Audio-Video Transport Working Group of the Internet Engineering Task Force (IETF) entwickelt wurde. Im Rahmen des Projektes wurden alle funktionalen und nicht-funktionalen Anforderungen umgesetzt und diese in einer Beispielanwendung integriert und getestet.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Was ist OHMComm?

OHMComm ist ein IT-Projekt der Informatik Fakultät an der technischen Hochschule in Nürnberg. Das Projekt wird im Rahmen des Bachelorstudienganges Informatik umgesetzt und erstreckt sich über zwei Semester. Herr Prof. Dr. M. Teßmann ist Projektinitiator und Projektleiter. Das Ziel des Projektes ist die Erstellung eines plattformunabhängigen Audiokommunikationsframeworks. Das Framework soll sämtliche Funktionalität, die für eine erfolgreiche Kommunikation zwischen zwei Teilnehmern benötigt wird, zur Verfügung stellen. Das Framework, als auch das Projekt, trägt den Namen OHMComm. Der Anwendungsfall für die Hochschule ist der Einsatz der Software in Lehrveranstaltungen, jedoch sind auch andere Anwendungsszenarien denkbar, wie z.B. die Integration in externe Softwareanwendungen.

1.2 Projektbeschreibung

Die Projektbeschreibung von Herr Prof. Dr. M. Teßmann beschreibt die Ziele, die im Rahmen des IT-Projektes erreicht werden sollten. Das Framework soll plattformunabhängig sein und Prinzipien des objektorientierten Designs entsprechen. Außerdem soll es über eine gute Dokumentation verfügen. Der obligatorische Funktionsumfang des Audiokommunikationsframework lässt sich in fünf größere Bereiche zerlegen. Zum einem wird für die Audioaufnahme und Wiedergabe eine Schnittstelle zur Audiohardware benötigt. Der zweite Block beinhaltet die Kodierung und Dekodierung von Audiodaten. Diese Funktionalität darf man keineswegs als optional betrachten, da in der Praxis eine unkomprimierte Audioübertragung meist nicht realisierbar ist. Die meisten Verbindungsleitungen verfügen hierfür nicht

über die notwendige Bandbreite. Das Versenden und Empfangen von Audiodaten mit dem Real-Time-Transport-Protokol (RTP) auf Basis des User Datagram Protocol (UDP) ist ein weiterer großer Implementierungsblock. RTP stellt ein standardisiertes Protokoll für die Audio- und Videoübertragung dar, welches 1996 von der Audio-Video Transport Working Group of the Internet Engineering Task Force (IETF) entwickelt worden ist [IETF, 2003]. Der vorletzte Block umfasst das Erstellen eines Jitterbuffers. Durch das Versenden von UDP-Paketen kann die Paketreihenfolge beim Empfänger nicht gewährleistet werden. Die Aufgabe des Buffers ist es, durch eine minimale Verzögerung vor dem Abspielen der Audiodaten, die Paketreihenfolge wiederherzustellen, wodurch sich die Audioqualität wesentlich erhöhen lässt. Der letzte Block beschäftigt sich mit den Themen Test und Bewertung. Sämtliche Funktionen des Frameworks sollen in einer Beispielanwendung validiert werden. Für eine objektive Bewertung des Framesworks sollen statistische Werte gesammelt werden können. Hierbei ist der Aspekt der Performanz zu berücksichtigen, da Latenzen eine wesentliche Rolle in der Audiokommunikation spielen.

1.3 Aufbau des Berichts

Der Aufbau des Berichts entspricht den klassischen Phasen der Softwareentwicklung und ist unterteilt in den Kapiteln Anforderungsanalyse, Entwurf, Implementierung, Tests und den Schlussbemerkungen. Das Kapitel Prototypische Voice-over-IP Konsolenanwendung entspricht dem Testkapitel. In der Anforderungsanalyse werden aus den Projektzielen die konkreten Anforderungen ermittelt und diese in funktionale und nicht-funktionale Anforderungen klassifiziert. Außerdem wird hier definiert, wie die Ziele und Anforderungen im Rahmen des Projektes interpretiert werden. In der Entwurfsphase werden für die Umsetzung der Anforderungen Lösungswege erarbeitet, verglichen und bewertet. Außerdem werden in diesem Abschnitt die einzelnen Komponenten des Frameworks spezifiziert sowie die Softwarearchitektur erstellt. Im Kapitel Implementierung wird der Plan aus der Entwurfsphase umgesetzt. Auf besonders wichtige Implementierungsdetails wird hingewiesen und diese erläutert. Die prototypische Voice-over-IP Konsolenanwendung wurde parallel neben den Framework entwickelt und dient ausschließlich zum Testen der Funktionalität. Da diese Anwendung auch ein Teil des Projektes ist, wird diese gesondert in einem eigenen Kapitel erörtert. Es wird erklärt wie die Funktionalitäten innerhalb der Anwendung getestet und umgesetzt worden sind. Im letzten Kapitel wird ein Fazit gezogen. Es wird überprüft, ob alle Ziele und Anforderungen umgesetzt sind. Außerdem gilt es zu klären, in welchen Umfang die Ziele erreicht wurden und wo es noch Optimierungsbedarf gibt. Im abschließenden Ausblick werden sinnvolle Erweiterungs- und Verbesserungsmöglichkeiten für das Framework vorgeschlagen.

Kapitel 2

Anforderungsanalyse

In diesem Kapitel sollen aus der Projektbeschreibung die daraus resultierenden konkreten Anforderungen für das Framework erstellt werden.

2.1 Funktionale Anforderungen

In diesem Abschnitt werden aus der Projektbeschreibung die funktionalen Anforderungen ermittelt und definiert. Die funktionalen Anforderungen werden in zusammenhängenden Gruppen eingeteilt.

- Schnittstelle zur Audiohardware

  Es muss eine entsprechende Schnittstelle zur Audiohardware, Mikrofone und Lautsprecher, erstellt werden. Diese wird benötigt, um die Audiodaten aufzunehmen und abzuspielen. Da ein Computer über mehr als ein Lautsprecher und Mikrofon verfügen kann, muss hier eine entsprechende Auswahlmöglichkeit existieren. Desweiteren müssen Konfigurationsmöglichkeiten vorhanden sein um z.B. bestimmte Abtastrate auszuwählen.

- Audiokompression

  Wie bereits erwähnt ist eine Audiokommunikation ohne entsprechende Datenkompression praktisch nicht realisierbar. Es muss eine allgemeine Schnittstelle für Dekodierer erstellt werden, die durch beliebige Audiocodecs erweitern lässt. Außerdem muss mindestens ein Codec integriert werden, um die Praxistauglichkeit des Frameworks zu gewährleisten.

- Netzwerkschicht

  Für den Verbindungsaufbau und -abbau sowie der Datenübertragung muss ebenfalls eine Schicht erstellt werden. Dabei müssen auch Verbindungsdaten verwaltet und verarbeitet werden. Die Übertragung basiert auf UDP.

- RTP-Protokoll

  Das Ent- und Verpacken von Nutzdaten in das RTP-Protokoll muss ebenfalls implementiert werden.

- Jitter-Buffer (RTPBuffer)

  Der Buffer hat die Aufgabe die Paketreihenfolge der empfangenen Pakete wiederherzustellen, da diese mit dem UDP-Protokoll nicht gewährleistet werden kann. Dadurch erhöht sich die Audioqualität. Der Buffer soll einen bestimmten Füllstand erreichen, bevor die Audiowiedergabe beginnt.

- Prototypische Voice-over-IP Konsolenanwendungen

  Diese Anwendung soll alle Funktionen des Frameworks testen. Dadurch kann sichergestellt werden, dass das Framework die gewünschte Funktionalität tatsächlich anbietet. Die Hauptaufgabe der Testanwendung ist es, die Kommunikation zwischen zwei Anwendern zu ermöglichen. Die Anwendung soll sich mit Parameter konfigurieren lassen.

- Statistik

  Um die Funktionen objektiv bewerten zu können, soll die Möglichkeit bestehen, statistische Werte zu erfassen und zu speichern.

2.2 Nicht-Funktionale Anforderungen

Hier werden aus der Projektbeschreibung die nicht-funktionalen Anforderungen ermittelt und definiert.

- Plattformunabhängigkeit

  Ist einer der zentralen Ziele des Frameworks. Es soll unter Windows, Linux und OS X funktionieren. Daraus folgt, dass eine Kommunikation zwischen einen Windows-Client und Linux-Client, oder einer sonstigen beliebigen Konstellation funktioniert, da die Funktionsweise immer identisch ist. Diesbezüglich wurde der Einsatz von CMake – ein Buildtool für C++ – festgelegt.

- Performanz

  Latenzen sollten für eine flüssige und störungsfreie Kommunikation so gering wie möglich gehalten werden. Daraus resultiert auch die Wahl der Programmiersprache C++. In der weiteren Entwicklungsphase gilt es diesen Aspekt mit erhöhter Priorität zu betrachten.

- Prinzipien des objektorientierten Designs

  Das Framework soll nach den Prinzipien des objektorientierten Designs entwickelt werden. Darunter versteht man Konventionen zur Komplexitätsreduzierung in der Softwareentwicklung, wodurch die Software wartungs- und änderungsfreundlicher wird [Lahres and Raýman, 2009][Kap. 3]. Das Framework soll modular aufgebaut sein, so dass die Komponenten auch einzeln nutzbar sind. Außerdem soll es möglich sein, die Klassen zu erweitern und auszutauschen. Konfigurationsmöglichkeiten sollen möglichst zur Laufzeit stattfinden.

- Dokumentation

  Das Erstellen einer Dokumentation ist ebenfalls eine Teilaufgabe des Projektes.

2.3 Übersicht aller Anforderungen

Es wird eine Übersicht aller Anforderungen mit kurzen Beschreibungen erstellt, um in den folgenden Kapiteln darauf referenzieren zu können.

a) Schnittstelle zur Audiohardware
   1. Erstellen der Schnittstelle
   2. Konfigurationsmöglichkeiten der Audiohardware

b) Audiokompression
   1. Erstellen der Schnittstelle
   2. Einbinden eines Audiocodecs

c) Netzwerkschicht
   1. Erstellen einer Netzwerkschicht
   2. Konfigurationsmöglichkeiten für Netzwerkverbindungen

d) RTP-Protokoll
   1. Implementierung des RTP-Protokolls

e) Jitter-Buffer
   1. Implementieren eines Jitter-Buffers
   2. Audiowiedergabe erst starten, wenn bestimmter Füllstand erreicht ist

f) Prototypische Voice-over-IP-Konsolenanwendung
   1. Basiert auf dem Framework und ermöglicht die Audiokommunikation zwischen zwei Anwendern
   2. Nimmt Parameter beim Programmstart entgegen und verarbeitet sie

g) Statistik
   1. Erfassen von statistischen Werten muss möglich sein
   2. Formatierte Ausgabe der Ergebnisse

h) Plattformunabhängigkeit
   1. Windows, Linux, OS X
   2. Einsatz von CMake

i) Performanz
   1. Performanz hat erhöhte Priorität

j) Prinzpien des objektorientierten Designs
   1. Erweiterbarkeit, Wiederverwendbarkeit, Austauschbarkeit der Komponenten

k) Dokumentation
   1. Erstellen einer Dokumentation

Kapitel 3

Entwurf

3.1 Projektverwaltung und Werkzeuge

In diesem Abschnitt werden die interne Projektverwaltung, die Werkzeuge zur Projektverwaltung und die Werkzeuge zur Projektrealisierung beschrieben. Die Aufgabenstellung und der Fortschritt des Projekts wurden fortlaufen in Treffen mit *Herrn Professor Teßmann* besprochen. Zwischen den Treffen stand *Herr Professor Teßmann* jederzeit für persönliche und schriftliche Fragen zur Verfügung.

3.1.1 GitHub

Zur kollaborativen Versionsverwaltung wird der Online-Service **GitHub** [GitHub, 2016] verwendet, welcher für OpenSource Projekte kostenlos öffentliche Repositories zur Verfügung stellt. Da das Projekt unter der **MIT-Lizenz** [OHMComm, 2016] entwickelt wird, ist die Nutzung eines öffentlichen Repositories ein Vorteil hinsichtlich auf eine mögliche Weiterentwicklung und einer guten Akzeptanz in der OpenSource-Community.

Das GitHub-Repository bietet die Möglichkeit, den Code, welcher lokal entwickelt wird, in das Repository hochzuladen, so dass immer ein konsistenter gleicher Versionsstand vorhanden ist. Außerdem gibt es die Möglichkeit, für einzelne Teilprojekte eigene **Branches** zu erstellen, so dass auf diesen unabhängig vom Master-Branch, in dem alle Teile zusammenfließen, entwickelt werden kann.

Zur Fehlerverwaltung wird das bereits Integrierte **Issue System** von GitHub verwendet. Mit diesem ist es möglich die Issues zu priorisieren, mit Tags zu versehen, einzelnen Projektmitgliedern zuzuweisen und den Bearbeitungsstand zu dokumentieren. Der weitere Vorteil hiervon ist, das jedes Projektmitglied automatisch eine

Benachrichtigung über neue Issues und Aktualisierungen erhält und somit bei der Fehleranalyse helfen kann. Außerdem wurden einzelne **Milestones** erstellt, welche sich an den fortlaufenden Besprechungen orientierten und somit laufend Ziele bei der Projektdurchführung definierten. Nach der Erreichung großer Projektziele wurden Versionsstände als **Releases** getaggt, so dass ersichtlich wurde, welchen funktionellen Umfang der Code in diesem Stadium hat.

3.1.2 CMake

Da das Ziel h des Projektes ist, plattformunabhängig zu sein, wird eine Möglichkeit benötigt unterschiedliche **integrierte Entwicklungsumgebungen (IDEs)** zu unterstützen. Hierfür wurde das plattformunabhängige Build Werkzeug CMake [CMake, 2016a], welches auch unter einer OpenSource Lizenz (BSD-Lizenz) [CMake, 2016b] erhältlich ist, verwendet. CMake erlaubt es für unterschiedliche IDEs Projektdateien zu erstellen, so dass für die Entwicklung nicht eine spezifische Entwicklungsumgebung benötigt wird, sondern mit mehreren, auch betriebssystemunabhängigen IDEs, entwickelt werden kann.

3.1.3 IDEs

Als IDEs kamen **Visual Studio** [Microsoft, 2015b] unter Windows 7 und Windows 10 zum Einsatz. Zu Beginn des Projekts in Version Visual Studio 2013, in der zweiten Hälfte des Projekts dann in Version Visual Studio 2015, welches eine bessere Unterstützung für C++11 Sprachfeatures enthält [Microsoft, 2015a]. Unter Fedora 21 wurde **Netbeans** [Oracle, 2015] in Version 8 eingesetzt.

3.1.4 Testumgebung

Zum Testen des Codes wurden mehrere unterschiedliche Testumgebungen eingesetzt. Dabei wurden **rechnerübergreifende Tests** des Verbindungsaufbaus und der Verbindungsqualität über *LAN*, *WLAN* und *WAN* mit verschiedenen Betriebssystemen durchgeführt.

Zum Testen einzelner Module des Projektes wurde die Unit-Test Umgebung **CPPTest** [Lundell, 2013] eingesetzt, die es ermöglicht einzelne Module auf ihre funktionalen Anforderungen hin zu überprüfen.

Für einen automatisierten Build-Test wurde auf den Online-Service **Travis CI** [TravisCI, 2016] gesetzt, der durch seine einfache Integration in das GitHub Repository eine gute Lösung bietet. Er bietet die Möglichkeit, schnell herauszufinden, ob der

neu eingecheckte Code fehlerfrei unter *Linux* und *OSX* kompiliert.

3.2 Build Prozess

3.2.1 CMake

Die CMake Konfiguration basiert auf mehreren `CMakeLists.txt`-Dateien, welche **hierarchisch** in die Projektstruktur integriert sind. In der Konfigurationsdatei auf der obersten Ebene wird die minimal benötigte CMake *Version* und der *Projektname* definiert. Außerdem werden IDE abhängige Einstellungen vorgenommen wie z.B. die Einstellung des *Warning Levels* für Compilefehler und die Unterstützung von *Unicode* Zeichen unter Visual Studio. Anschließend werden die relativen Pfade, in welche die fertig kompilierten Dateien abgelegt werden, definiert und die unteren Ordnerstrukturen mit ihren dort befindlichen Konfigurationsdateien angegeben.

In diesen weiteren Konfigurationsdateien für **OHMComm** und den extern benutzten Softwarekomponenten wie **Rtaudio** und **Opus** wird nun festgelegt, welche *Header* und welche *C++ Dateien* eingebunden werden.

3.2.2 Build unter Unix

Zum Erstellen des Projektes OHMComm unter Unix-Systemen wird neben einem C++11-fähigen Compiler und dem Programm **CMake** noch die **Make** Build-Suite sowie eine Audio-Bibliothek mitsamt Header-Dateien benötigt. Während Make und eine Audio-Bibliothek auf den meisten Unix-Systemen bereits mitgeliefert werden, müssen die Entwicklungs-Header für die verwendete Audio-Bibliothek meist erst noch installiert werden. Als Audio-Bibliothek wird unter Linux-Systemen OSS, ALSA, Jack und PulseAudio unterstützt [Scavone, 2014b]. Die Entwicklungs-Header für die Audio-Bibliotheken liegen je nach System – oder genauer: je nach Paketverwaltungssystem – in verschieden benannten Paketen. So heißt das Paket für die ALSA-Header unter Debian `libasound-dev` und unter Fedora `alsa-lib-devel`. Unter Linux und den meisten Unix-Betriebssystemen wird das Projekt kompiliert, indem zuerst mit dem Befehl `cmake -G "Unix Makefiles"` aus der CMake-Beschreibung eine `Makefile` – also eine Anleitung für das `make` Build-System – erstellt wird. Daraufhin können mit dem Befehl `make` im Zielverzeichnis des vorherigen Kommandos die einzelnen Bibliotheken oder Programme kompiliert werden. So erstellt `make OHMCommLib` die Bibliothek **OHmCommLib** (das eigentliche Framework), die in weitere Programme eingebunden werden kann (siehe Abschnitt 3.3.1). `make OHMComm` erstellt das ausführbare Programm **OHMComm** (siehe Ziel f) und `make Tests` die Test-Suite für das Projekt.

3.2.3 Build unter Windows

Zum Erstellen des Projekts unter Windows wird eine aktuelle Version von **CMake** und die zu benutzende **Entwicklungsumgebung**, welche von CMake unterstützt wird, benötigt. Für einen sauberen Entwicklungsprozess ist es zu empfehlen, sich ein Build Verzeichnis außerhalb des Sourcecodes anzulegen (Out-of-Source Build).

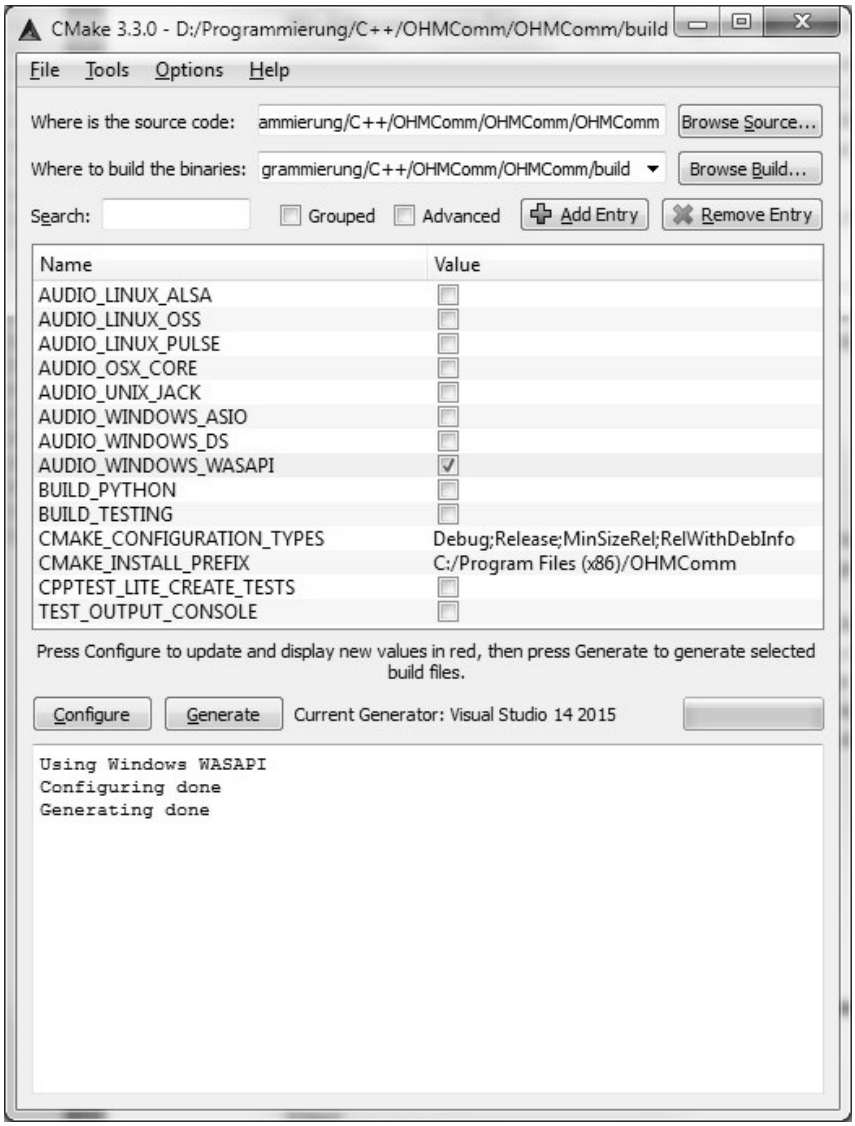


Abbildung 3.1: CMake Konfiguration unter Windows 7

Es wird CMake mitgeteilt wo sich der Source Code (`Browse Source`) und das Build Verzeichnis (`Browse Build`) befindet. Mittels `Configure` wird der Compiler und die verwendete Entwicklungsumgebung definiert. Anschließend wird ausgewählt welche Audio Bibliothek verwendet wird, unter Windows wird *ASIO*,

DS und *WASAPI* unterstützt. Wahlweise kann auch *CppTest* aktiviert werden, falls die Tests erstellt werden sollen. Die Einstellungen werden durch nochmaligen Druck auf `Configure` bestätigt und über `Generate` werden die Projektdateien im vorher definierten Buildverzeichnis erstellt.

Es wird die *Entwicklungsumgebung* mit den generierten Projekt Dateien gestartet und angewiesen das Projekt zu kompilieren.

Im Verzeichnis `build/Release/` findet sich bei der Verwendung von Visual Studio und Kompilierung mit Release Optionen das ausführbare Programm **OHMComm.exe** und die OHMComm Bibliothek **OHMComm.lib**.

3.3 Softwarearchitektur

3.3.1 Konfiguration und Verwendung

Um OHMComm verwenden zu können, müssen vorher eine Vielzahl an Einstellungen getroffen werden. Diese Einstellungen gliedern sich in folgende Bereiche:

Audio-Konfiguration: Einstellungen für die Soundkarte, wie die Wahl des Formats zum Aufnehmen oder Abspielen, die Anzahl der Kanäle, die Abtastrate oder die verwendeten Audiogeräte. Die Auswahl der Geräte für die Aufnahme und Ausgabe können unabhängig vom Kommunikationspartner eingestellt werden. Andere Einstellungen (Audioformat, Kanäle und Abtastrate) müssen jedoch mit dem Gesprächspartner abgestimmt oder auf kompatible Werte umgerechnet werden.

Prozessoren-Konfiguration: Hierunter fällt die Auswahl der verwendeten Audioprozessoren und die Reihenfolge, in der die Prozessoren verkettet werden (siehe Abschnitt 3.3.3). Ebenso besitzen manche Prozessoren eigene Einstellungsmöglichkeiten, um deren Funktionsweise zu regeln. Die meisten Prozessoren und Prozessoreinstellungen fordern keine Anpassung der Konfiguration des Gesprächspartners. Ausnahmen sind hier die Audiocodecs, die von beiden Programmen gleich konfiguriert verwendet werden müssen um die encodierten Daten wieder richtig decodieren zu können.

Netzwerk-Konfiguration: Bestimmt den zu verwendenden Port zum Empfangen und Senden von Paketen auf dem lokalen Rechner, sowie die IP-Adresse und den Port des Rechners des Kommunikationspartners. Die Ports und die Adresse des jeweils anderen Rechners müssen vorher zwischen den Gesprächspartner abgestimmt werden, um eine Duplex-Kommunikation einrichten zu

können. Bei der Netzwerk-Konfiguration ist zu beachten, dass bei Kommunikation über ein WAN (Wide Area Network) eine von außen erreichbare IP-Adresse gewählt wird evtl. auch Port-Weiterleitungen eingerichtet werden müssen.

Sonstige Konfiguration: Hier zählen sonstige, rein optionale Einstellungen, die die eigentliche Audiokommunikation nicht beeinflussen, wie das Messen der Ausführungsdauer der verwendeten Prozessoren, das Schreiben des Logs in eine Datei sowie die informativen Daten, die bei RTCP SDES-Paketen gesendet werden (siehe Abschnitt 3.3.5). Da diese Einstellungen nur das lokale Programm betreffen, müssen sie nicht mit dem Gesprächspartner abgestimmt werden.

Für die meisten nicht-optionalen Einstellungen sind Standardwerte vorgegeben (wie die beiden Ports für den lokalen und entfernten Rechner) oder werden beim Start des Programms ermittelt (wie die Standard-Audiogeräte für die Ein- und Ausgabe). Um die einfachste Form der Kommunikation aufbauen zu können – ohne Audiocodecs oder sonstigen Audioprozessoren – muss nur die IP-Adresse des Gegenübers gesetzt werden. Jedoch empfiehlt es sich aus verschiedenen Gründen (wie die Reduzierung der Bandbreite) eine erweiterte Konfiguration vorzunehmen.
OHMComm implementiert eine Vielzahl an Konfigurationsmöglichkeiten, um einen möglichst breiten Verwendungsbereich zu bieten. So kann die prototypische Anwendung aus Kapitel 5 als interaktive Konsolenanwendung gestartet werden. Dabei werden alle Einstellungsmöglichkeiten nacheinander ausgegeben und der Benutzer kann durch Eingabe einen der vorgeschlagenen Werte auswählen oder einen eigenen Wert eingeben, je nach Art der Einstellung.
Ebenso kann die Konfiguration durch Kommandozeilen-Argumente vorgenommen werden. Hierfür benutzt OHMComm die bekannte Syntax aus Unix, bei der Schlüssel-Wert Paare mit einem Gleichheitszeichen getrennt angegeben werden, z.B. `-local-port=54321` für die Bestimmung des lokalen Ports. Ebenso werden für die meisten Optionen sowohl ein kurzer als auch ein langer Schlüssel unterstützt. So geben beide Argumente `-h` und `-help` die Hilfe auf der Kommandozeile aus, die alle verfügbaren Parameter und deren Bedeutung sowie Standard-Werte anzeigt. Die gleichen Parameter können auch aus einer Konfigurationsdatei geladen werden. Dafür werden dort die Schlüssel-Wert Paare zeilenweise und auch durch ein Gleichheitszeichen getrennt (aber ohne führende Bindestriche) aufgelistet und die Datei beim Programmstart als einziger Parameter übergeben.
Um OHMComm auch als Bibliothek verwenden zu können, wird eine Möglichkeit geboten über Methodenaufrufe die benötigten und optionalen Einstellungen zu setzen. Somit kann das Framework sehr einfach in ein Programm integriert werden und von dort aus auch konfiguriert werden.
Des Weiteren gibt es die sog. **passive Konfiguration**, bei der alle Konfigurationen,

die in beiden Programmen gleich eingestellt sein müssen, vor dem Start der Kommunikation ausgetauscht werden. Zu den ausgetauschten Einstellungen zählen Abtastrate, Audioformat, Anzahl der Kanäle und die verwendeten Prozessoren (für die Audiocodecs). Somit wird die Gleichheit dieser Einstellungen garantiert und der Konfigurationsaufwand verringert. Mehr Informationen zur Umsetzung der passiven Konfiguration finden sich in Abschnitt 4.6.

3.3.2 Audio-Schnittstelle

Für die Audioverarbeitung ist ein allgemeine Schnittstelle zur Audiohardware nötig, siehe Anforderung a. Dabei gilt es insbesondere die nicht-funktionalen Anforderungen h, i und j zu berücksichtigen. Es wird eine abstrakte Klasse mit dem Namen `AudioHandler` erstellt, welche die Verbindung zur Hardware darstellt. Sämtliche Audiodaten laufen über diese Schnittstelle zum Mikrofon oder zu den Lautsprechern.

Verarbeitungsmethoden

Die Klasse bietet drei virtuelle Methoden, welche eine automatische Verarbeitung starten. Die Methode `startRecordingMode()` startet die Verarbeitung von Audioinputdaten (Mikrofon), `startPlaybackMode()` startet die Verarbeitung von Audiooutputdaten (Lautsprecher) und `startDuplexMode()` ist die gleichzeitige Verarbeitung von Audioinput- und Audiooutputdaten. Letztere stellt den Standardfall für die Audiokommunikation dar. Wie diese Methoden im Detail aussehen, wird in Unterklassen von `AudioHandler` implementiert. Für die Hardwareansteuerung ist es sinnvoll, bereits existierende Lösungen zu verwenden. In diesem Fall kann man Unterklassen von `AudioHandler` auch als Wrapper-Klassen ansehen.

Konfiguration

Vor der Verarbeitung muss die Hardware konfiguriert werden. Hierfür muss das Struct `AudioConfiguration` verwendet werden. In diesem lassen sich das Mikrofon und die Lautsprecher bestimmen, die Anzahl der Kanäle (Mono - Stereo), die Sample Rate und die Größe des internes Buffers. Der interne Buffer kann vorerst vernachlässigt werden. Im Kapitel von 3.4.1 wird dieser näher erläutert. Für das Setzen der Einstellungen steht im `AudioHandler` die Methode `setConfiguration()` zur Verfügung. Falls keine Konfiguration übergeben wird, so wird automatisch die virtuelle Methode `setDefaultAudioConfig()` aufgerufen. Diese muss ebenfalls von der Unterklasse implementiert werden. Ihre Aufgabe ist es eine Standardkonfi-

guration zur Verfügung zu stellen, falls die Klasse nicht konfiguriert wurde. Es ist jedoch zu beachten, dass Standardkonfiguration eventuell fehlerhaft und nicht auf allen Systemen lauffähig ist.

Steuerung der Verarbeitung

Eine laufende Verarbeitung kann durch folgende Methoden gesteuert werden: Die Methode `suspend()` pausiert die aktuelle Verarbeitung und `resume()` setzt sie weiter fort. Die Methode `stop()` bricht den gesamten Verarbeitungsvorgang ab, welcher anschließend auch nicht mehr mit `resume()` fortgesetzt werden kann. `reset()` ruft intern `stop()` auf und setzt die übergebene Audiokonfiguration zurück.

Prepare-Methode

Die virtuelle Methode `prepare()` nimmt eine Sonderrolle ein. Grundsätzlich kann Sie als optional betrachtet werden, jedoch ist sie für bestimmte Einsatzzwecke sinnvoll. Sie sollte aufgerufen werden nachdem eine Audiokonfiguration übergeben worden ist, jedoch bevor die Verarbeitung gestartet wird. Dies kann sinnvoll sein um z.B. die ausgewählte Hardware im Hinblick auf bestimmte Einstellungsmöglichkeiten und deren Kompatiblität mit anderen Komponenten zu überprüfen. Falls eine Inkompatiblität festgestellt wird, kann der Start der Verarbeitung abgebrochen werden.

Fazit

Abbildung 3.2 zeigt eine vereinfachte Darstellung des `AudioHandlers`. Durch die abstrakte Klasse wird gewährleistet, dass die Klasse austauschbar und erweiterbar ist. Viele Implementierungsdetails werden in die Unterklasse ausgelagert. Durch das Verwenden externer Libraries kann die Unterklasse auch als Wrapper-Klasse angesehen werden. Die Audiohardware lässt sich durch das Struct `AudioConfiguration` konfigurieren. Jedoch besteht diese Klasse nicht nur aus der Schnittstelle zur Hardware, sondern ist der zentraler Verarbeitungspunkt, daher einer der wichtigsten Komponenten des Frameworks. Für die Verarbeitung von Audiodaten wurde eine spezielle Schnittstelle, die Verarbeitungskette, integriert. Im folgenden Kapitel wird diese näher erläutert.

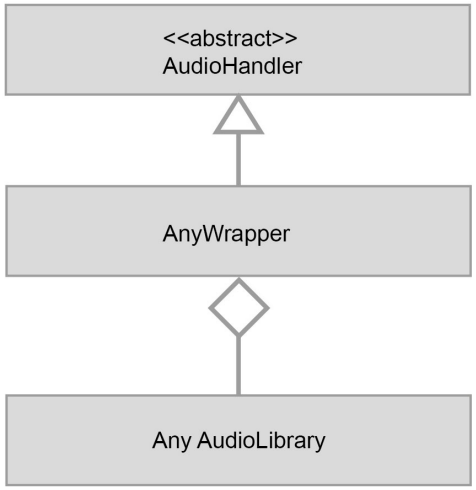


Abbildung 3.2: UML-Klassendiagramm des AudioHandlers

3.3.3 Verarbeitungskette

Die Klasse `AudioHandler` bietet eine Schnittstelle zur Hardware an. Ein Teil ihrer Implementierung ist die Verarbeitungskette. Durch ihr können andere Klassen mit den Audiodaten arbeiten, ohne die Implementierungsdetails des `AudioHandlers` kennen zu müssen. Im folgenden Abschnitt wird erklärt wie diese funktioniert.

AudioProcessor

Eine Klasse, die mit Audiodaten arbeiten will, muss das Interface `AudioProcessor` implementieren, siehe Abbildung 3.3. Hierfür muss sie die Methoden `processInputData()` und `processOutputData()` implementieren. `processInputData()` wird immer aufgerufen, wenn Audiodaten vom Mikrofon für die Verarbeitung im internen Buffer vorhanden sind. Die Methode `processOutputData()` wird aufgerufen, wenn die Soundkarte Zeit hat etwas abzuspielen. Dort können Daten auf den Buffer zum Abspielen hinterlegt werden. Beide Funktionen haben als Parameter den jeweiligen Buffer und die Größe des Buffers. Der Rückgabewert der Funktion ist die neue oder alte Größe des Buffers. Der Rückgabewert ist nur auf Hinblick des Dekodieren und Kodieren relevant.

`configure()` und `cleanUp()` sind weitere Methoden des Audioprozessors, die nicht in der Abbildung 3.3 zu sehen sind. Erstere wird einmalig aufgerufen, kurz bevor die Verarbeitung gestartet wird. Als Parameter muss die festgelegte Audiokonfiguration übergeben werden. Die Funktion hat einen boolschen Rückgabewert der angibt, ob der Prozessor bereit für die Audioverarbeitung ist. Im Fehlerfall star-

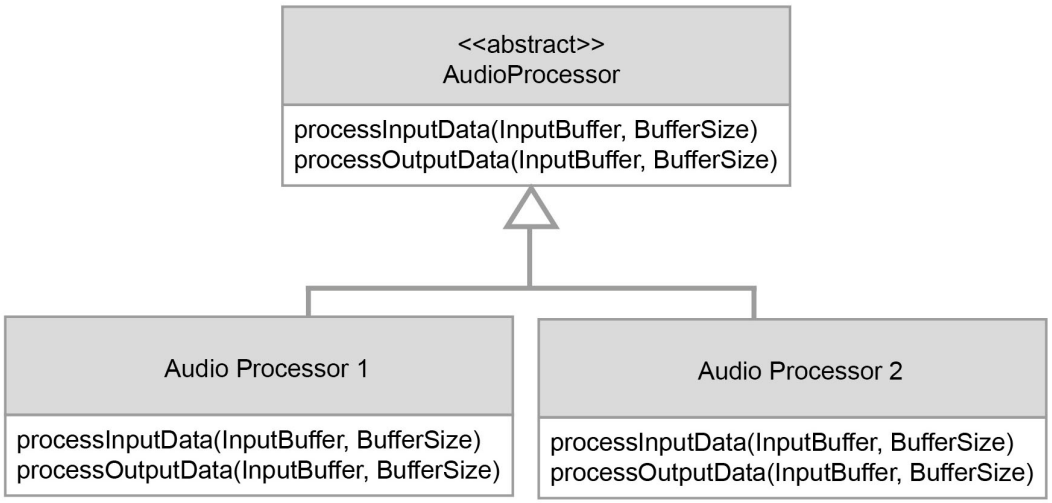


Abbildung 3.3: Beispielhafte Verwendung der AudioProcessor-Klasse

tet die Verarbeitung nicht. Dies kann wichtig sein, wenn z.B. ein `AudioProcessor` eine gewisse Audiokonfiguration voraussetzt. `cleanUp()` dient dazu, den `AudioProcessor` ordentlich aufzuräumen, wenn dieser nicht mehr benötigt wird.

Registrieren von Audioprozessoren

Das Implementieren der Klasse `AudioProcessor` reicht nicht aus, um an der Audioverarbeitung aktiv teilzunehmen. Abbildung 3.4 zeigt die Beziehung zwischen `AudioHandler` und `AudioProcessor`. Die Klasse muss sich beim `AudioHandler` mit der Methode `addProcessor()` anmelden. Diese nimmt als Parameter einen `AudioProcessor` entgegen. Audioprozessoren lassen sich über ihren Namen eindeutig identifizieren. Der Prozessor wird, falls er nicht schon vorhanden ist, in einer Liste vom Typ `AudioProcessor` eingefügt. Falls nun Audioinputdaten anliegen, so werden von allen Audioprozessoren, die sich in der Liste befinden, die `processInputData()`-Methoden aufgerufen. Die Aufrufreihenfolge ist abhängig von der Anmeldereihenfolge. Für Audiooutput-Daten geschieht das analog, jedoch ist hier die Aufrufreihenfolge entgegengesetzt, da die Verarbeitungskette nicht kommutativ ist. Nur so ist es möglich an die ursprünglichen Audiodaten zu gelangen. Das Konzept hat einige Vorteile, z.B. können sich beliebige viele Prozessoren beim `AudioHandler` anmelden, jedoch ist hierbei der Aspekt der Performanz zu berücksichtigen. Des weiteren können Audioprozessoren zur Laufzeit hinzugefügt werden und beeinflussen sich nicht gegenseitig. Änderungen am `AudioHandler` sind ebenfalls unabhängig von den Prozessoren. Vom Aufbau ähnelt die Verarbeitungskette dem Observer-Pattern, jedoch mit dem entscheidenden Unterschied,

dass die Einfügereihenfolge eine wesentliche Rolle für die Verarbeitung spielt.

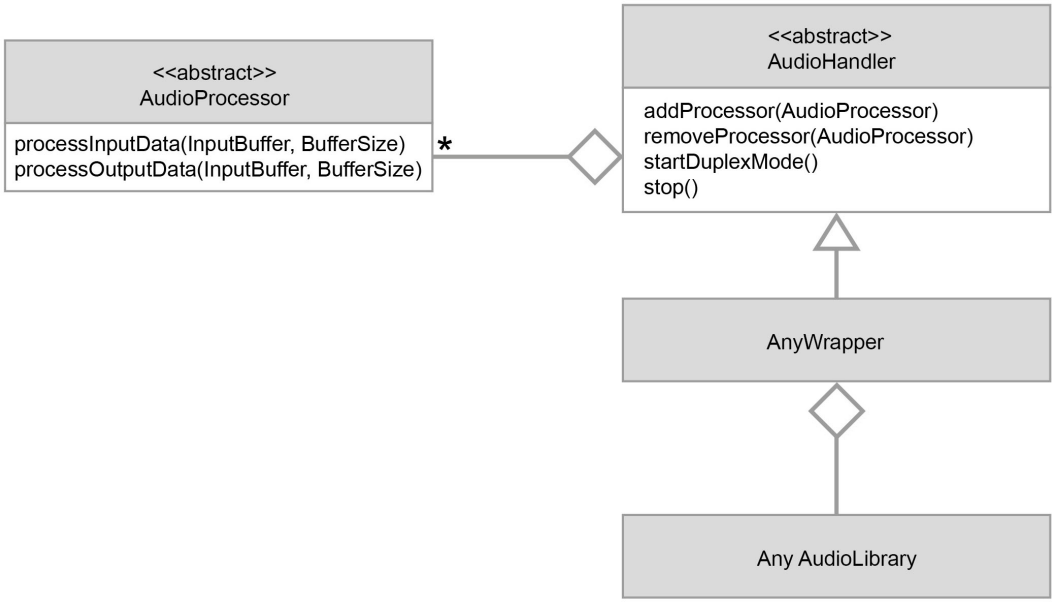


Abbildung 3.4: UML-Klassendiagramm AudioHandler und AudioProcessors

ProfilingAudioProcessor

Beim `ProfilingAudioProcessor` handelt es sich um eine spezielle Unterklasse des `AudioProcessors`, siehe Abbildung 3.5. Der Prozessor hat die Aufgabe, Anforderung g umzusetzen. Hierzu wird das Decorator-Pattern eingesetzt, welches ermöglicht ein Objekt dynamisch zur Laufzeit zu erweitern. Der `ProfilingAudioProcessor` ist dabei zum einem Unterklasse des zu erweiterten Objekt und zum anderen hat es eine Referenz darauf. Der Konstruktor des `ProfilingAudioProcessor` nimmt als Parameter einen Pointer vom Typ `AudioProcessor` entgegen und speichert die Referenz. Der `ProfilingAudioProcessor` muss die Methoden der Oberklasse implementieren, jedoch werden deren Aufrufe an die gespeicherte Referenz des `AudioProcessors` weitergeleitet. Funktionen die erweitert werden sollen, können vor und nach dem Weiterleitungsaufruf ergänzt werden.

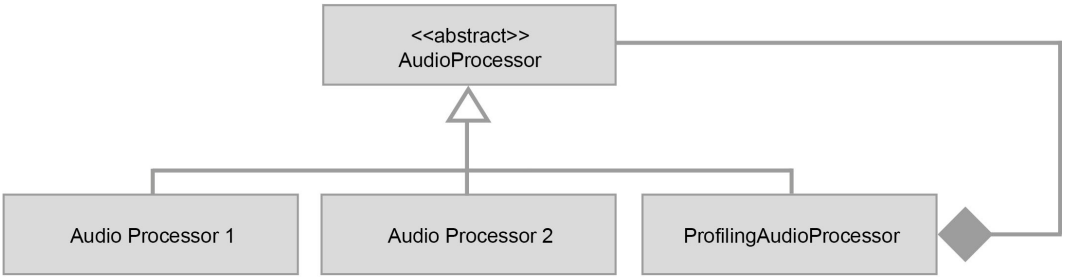


Abbildung 3.5: UML-Klassendiagramm des ProfilingAudioProcessors

3.3.4 Austauschbarkeit und Instantiierung

Um der Anforderung j gerecht zu werden und die Komponenten austauschbar zu halten, wird das Entwurfsprinzip `Program to an interface, not to an implementation` [Lahres and Raýman, 2009][Kap. 3.5] umgesetzt. Dies ermöglicht eine höhere Flexibilität, da der abstrakte Typ zur Laufzeit durch ein konkretes Objekt ausgetauscht werden kann. In diesem Zusammenhang bietet es sich an das Factory-Method-Pattern zu integrieren. Dabei wird die Instantiierung in einer Fabrikmethode ausgelagert. Als Parameter erwartet die Funktion ein String mit dem Namen der zu instantiierten Klasse. Alle Klassen basieren auf den gemeinsamen abstrakten Basistypen, der auch gleichzeitig Rückgabetyp ist. Die Fabrikmethoden wurde in eigenen Fabrikklassen ausgelagert, wie in Abbildung 3.6 zu sehen ist. `getAudioHandler()` und `getAudioProcessor()` stellen die Fabrikmethoden dar. `getAudioHandlerNames()` und `getAudioProcessorNames()` liefern die Namen der Klassen, die instantiierbar sind.

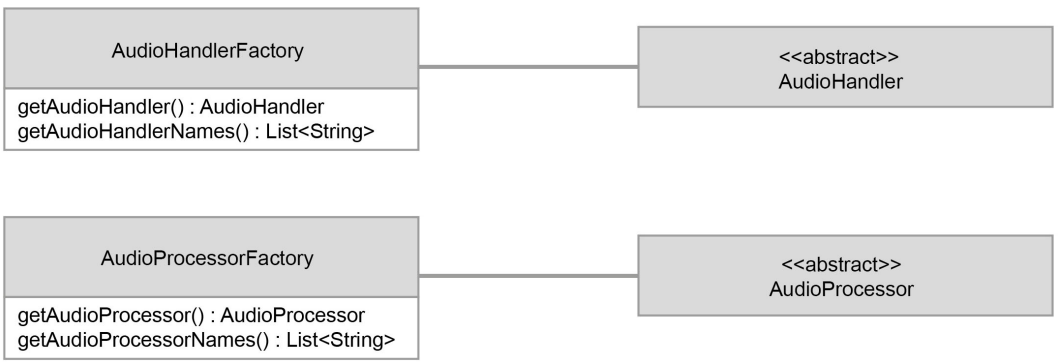
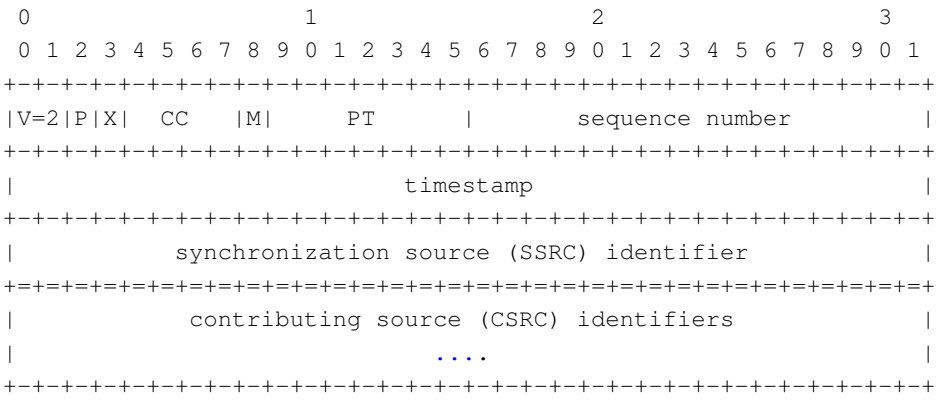


Abbildung 3.6: Factory-Klassen im Überblick

3.3.5 RTP-Protokoll

Das Real-time Transport Protocol (RTP) ist ein Netzwerkprotokoll für die Übertragung von Echtzeitdaten, wie Audio- und Videostreams oder -Konversationen. Es ist im RFC 3550 der Internet Engineering Task Force (IETF) definiert (siehe [IETF, 2003]) und unterstützt sowohl Unicast- als auch Multicast-Sitzungen. RTP ist ein Protokoll für die Anwendungsebene und kann auf beliebigen Transportprotokollen wie UDP oder TCP aufgesetzt werden. Jedoch wird RTP meist mit UDP verwendet, da aufgrund der Echtzeitvoraussetzungen der Verlust von Paketen eher geduldet werden kann als das Blockieren der Kommunikation durch erneutes Senden nicht-angekommener Pakete, so wie es in TCP üblich ist. Dies hat wiederum zur Folge, dass eine auf RTP mit UDP basierende Anwendung den Verlust einzelner Pakete kompensieren muss. Ein RTP-Paket besteht aus dem RTP-Header und dem anwendungsspezifischen Payload (Body). Der RTP-Header hat eine Größe von zwölf bis 72 Bytes und definiert folgende Felder:



Listing 3.1: RTP Header [IETF, 2003]

Version: Hat in RFC 3550 immer den Wert 2, frühere Versionen hatten den Wert 1.

Padding-Bit: Gibt an, ob die Daten des RTP-Pakets auf ein Vielfaches von 4 Byte gepadded sind.

Extension-Bit: Gibt an, ob eine RTP Header-Extension existiert, die direkt an den Header anschließt.

CSRC Count: Gibt die Anzahl der Contribution Sources (CSRCs) an, maximal 15 aufgrund der Größe des Feldes mit 4 Bit.

Marker-Bit: Anwendungsspezifische Bedeutung.

Payloadtype: Gibt den Typ der transportierten Daten an. Dieser kann aus einer Liste vordefinierter Typen aus RFC 3551 oder dynamisch ausgewählt sein.

Sequence number: Gibt die Position des RTP-Pakets innerhalb des Datenstroms an und wird für die Umsortierung der Pakete verwendet. Der Anfangswert dieses Felds sollte zufällig gewählt werden.

Timestamp: Zeitpunkt, zu dem das erste Byte des Pakets erstellt wurde, wobei auch hier der Anfangswert zufällig gewählt wird.

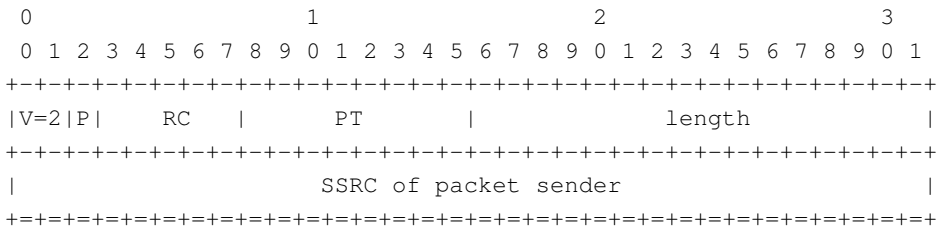
Synchronization Source Identifier: Zufällig gewählte Zahl zum eindeutigen Identifizieren des Senders dieses Pakets, auch SSRC genannt.

Contribution Source Identifier: Beinhaltet die originalen SSRCs der Teilpakete, wenn das Paket von einem Mixer aus verschiedenen Teilpaketen zusammengestellt wurde, werden auch CSRCs abgekürzt.

Im OHMComm-Framework wird RTP auf Basis von UDP verwendet. RTP wird hierbei verwendet, um den Verlust von Paketen zu Entdecken und eine Sortierreihenfolge festzulegen. Da OHMComm keine Sitzungen mit mehr als zwei Teilnehmern oder verschiedene Payload-Typen innerhalb einer Sitzung unterstützt, haben die meisten Header-Felder für das Framework keine Bedeutung. Jedoch werden für gesendete RTP-Pakete alle Header-Felder richtig belegt, um eine Interoperabilität mit anderen VoIP-Programmen zu gewährleisten. Für das Auslesen und Erstellen von RTP-Paketen sowie Generieren der richtigen Header-Felder ist die Klasse `RTPPackageHandler` verantwortlich. So könnte man z.B. mithilfe von OHMComm eine Sitzung mit mehreren Teilnehmern aufbauen, indem der lokale Client mit einem RTP-Mixer kommuniziert, der die Pakete dann auf die anderen Teilnehmer verteilt und auch deren Pakete vereint.

RTCP-Protokoll

Das Real-time Transport Control Protocol (RTCP) ist auch in RFC 3550 definiert und bietet Möglichkeiten, Flusskontrolle für eine RTP-Sitzung durchzuführen sowie statistische Daten über die Verbindungsqualität (oder Quality of Service, kurz: QoS) zu übertragen. Ebenso werden Pakettypen für anwendungsspezifische Daten und zum Beenden einer RTP-Sitzung bereitgestellt. Im Gegensatz zu RTP-Paketen beinhalten RTCP-Pakete keine Daten, sondern bestehen nur aus RTCP-Headern. Jedoch ist in RFC 3550 vorgegeben, dass ein RTCP-Header nie einzeln, sondern nur in Gruppen von mindestens zwei Headern, sogenannten „compound packages“, versendet wird. Die verschiedenen RTCP-Pakettypen bestehen aus den ersten acht Bytes, die für alle Typen die gleiche Bedeutung haben, sowie einen Typ-spezifischen Teil. Der gemeinsame Header-Teil besteht aus den folgenden Feldern:



Listing 3.2: RTCP Header [IETF, 2003]

Version: Hat in RFC 3550 immer den Wert 2, frühere Versionen hatten den Wert 1.

Padding-Bit: Gibt an, ob der RTCP-Header auf ein Vielfaches von 4 Byte gepadded ist. Padding darf nur das letzte Teilpaket eines zusammengesetzten Pakets angewendet werden.

Item Count: Gibt die Anzahl der Einträge in diesem RTCP-Header an. Die Bedeutung hiervon ist abhängig vom Pakettyp

Package Type: Gibt den Typ des RTCP-Paketes an

length: Die Länge des RTCP-Teilheaders in 32-Bit Blöcken, minus 1 (den ersten 32-Bit Block)

SSRC: Source Description des Senders, stimmt mit der SSRC von RTP-Paketen dieses Senders überein

RFC 3550 definiert folgende RTCP Pakettypen und deren Inhalten:

Sender Report: Beinhaltet Informationen über die Anzahl an gesendeten Paketen und Daten eines Senders. Ebenso werden pro Teilnehmer, von dem der Sender des Sender Reports Pakete empfangen hat, Daten über die Quality of Service (Anzahl verlorener Paketen, Netzwerkjitter, letzter empfangener Sender Report) verschickt. Diese Reception Report genannten Blöcke ermöglichen, dass jeder Teilnehmer einer RTP-Sitzung von jedem anderen Teilnehmer in regelmäßigen Abständen Feedback über die Übertragungsqualität bekommt. Jeder aktive Sender in einer RTP-Sitzung muss alle zusammengesetzten RTCP-Pakete immer mit einem Sender Report beginnen.

Receiver Report: Äquivalent zum Sender Report für passive Teilnehmer (also Teilnehmer die selbst keine Daten verschicken). Beinhaltet keine Informationen zur Anzahl verschickter Daten, aber trotzdem die Liste der Reception Reports für alle anderen Sender. Jeder passive RTP-Teilnehmer muss ein zusammengesetztes Paket immer mit einem Receiver Report beginnen.

Source Description: Beinhaltet eine Liste an informativer Daten über den Teilnehmer mit der angegebenen SSRC. Diese Daten, wie Telefonnummer, E-Mail, Webseite und Name können von Client-Programmen für die anderen Teilnehmer einer Sitzung angezeigt werden.

BYE: Der BYE RTCP-Header benachrichtigt die Teilnehmer einer Sitzung, dass der Sender mit der angegebenen SSRC die Sitzung verlässt und muss immer als letztes Paket eines zusammengesetzten Pakets stehen.

Application-defined: Der letzte Pakettyp bietet die Möglichkeit, anwendungsspezifische Daten zu übertragen. Dafür wird ein Name zur Identifikation des Typs der Daten und ein variabel großer Block für die eigentlichen Daten bereitgestellt.

Im OHMComm-Framework werden RTCP-Pakete in der Klasse `RTCPPackage-Handler` erstellt sowie ausgelesen und in `RTCPHandler` verschickt und empfangen. `RTCPHandler` besitzt einen eigenen Thread, der auf einem dedizierten RTCP-Port auf ankommende Pakete wartet sowie in regelmäßigen Abständen (oder wenn explizit dazu aufgefordert) RTCP-Pakete verschickt. Folgende Pakettypen werden von der RTCP-Implementierung verwendet:
In regelmäßigen Abständen wird ein zusammengesetztes Paket aus **Sender Report** (mit einem Reception Report) und **Source Description** versendet, wie es RFC 3550 vorschreibt. Jede empfangenen Sender Report oder Source Description Pakete werden in der prototypischen Anwendung derzeit nicht weiter verarbeitet, sondern nur auf der Konsole ausgegeben. Wenn die OHMComm-Anwendung beendet wird, wird zu den beiden genannten Pakettypen ein **BYE**-Paket angehängt, das auf der Empfänger-Seite dafür sorgt, dass die Kommunikation auch dort ordnungsgemäß abgebaut wird. Somit wird beim Beenden eines der Klienten der andere auch beendet, zumindest für die Kommunikation zwischen zwei OHMComm-Frameworks. Zusätzlich wird für die **passive Konfiguration** aus Abschnitt 3.3.1 der **Application-defined** Pakettyp verwendet, um eine Konfigurationsanfrage zu stellen und die geteilten Einstellungen auszutauschen. Der genaue Ablauf der passiven Konfiguration wird in Abschnitt 4.6 beschrieben.

3.3.6 Jitter-Buffer

Wie bereits in Abschnitt 3.3.5 erwähnt, wird eine Echtzeitübertragung mit RTP meist mit dem Transportprotokoll UDP verwendet. Da UDP aber nicht garantiert, dass versendete Pakete beim Empfänger ankommen und auch nicht, in welcher Reihenfolge, muss die Anwendung dafür sorgen, dass mit verlorenen Paketen oder Paketen, die in der falschen Reihenfolge empfangen werden, richtig umgegangen wird. Dafür

gibt es auf der Empfängerseite einen Jitter-Buffer, einen Puffer, der empfangene Pakete speichert und aus dem die weitere Prozessorkette Pakete in der richtigen Reihenfolge auslesen kann. Der Begriff Jitter-Buffer kommt von dem englischen Wort Jitter, dass für die Netzwerkverzögerung, – oder genauer: die Varianz der Netzwerkverzögerung – steht. Wie der Name schon andeutet, ist es eine der Aufgaben eines Jitter-Buffers, die Verzögerung des Netzwerkes und deren Schwankung auszugleichen. Ebenso sortiert ein Jitter-Buffer die empfangenen Pakete anhand ihrer RTP Sequenz-Nummer um und kaschiert den Verlust von Paketen. Bei Bibliotheken oder Programmen, die Sitzungen mit mehreren Teilnehmern unterstützen – was bei OHMComm nicht der Fall ist – muss für jeden anderen Sender ein eigener Jitter-Buffer verwaltet werden, da die Pakete der verschiedenen Sender verschiedene Netzwerkverzögerungen aufweisen können.
Bei der Netzwerkübertragung über UDP können Pakete verloren gehen, oder sie werden so spät empfangen, dass sie bereits hätten abgespielt werden müssen, sog. „late loss“. Um den Audiotreiber trotzdem Daten zum Abspielen zu liefern, und nicht die Ausgabe ins Stocken zu bringen, muss ein Echtzeitkommunikationsprogramm dafür sorgen, dass diese Verluste von Audiodaten ausgeglichen werden. Diese Funktion nennt sich „loss concealment“ (also: Verstecken von Verlusten) und kann auf verschiedene Arten umgesetzt werden. Die einfachste Möglichkeit ist es, wenn ein Paket angefordert wird, dass (noch) nicht empfangen worden ist, die Audiodaten dieses Pakets mit **Stille** zu ersetzen. Stille lässt sich sehr einfach erzeugen (z.B: durch Setzen aller Samples auf Null), ist ab einer gewissen Dauer für den Menschen hörbar und unterbricht den Fluss des Gesprächs, das Gespräch kling abgehakt. Als weitere Möglichkeit kann anstatt der Stille ein zufällig generiertes leises Rauschen, sog. „**comfort noise**“, abgespielt werden. Im Gegensatz zur absoluten Stille bekommt man bei Rauschen nicht so schnell das Gefühl, dass die Verbindung abgebrochen ist. Eine dritte Möglichkeit wiederholt das letzte erfolgreich empfangene Paket und spielt es erneut ab. Da sich der Tonverlauf der menschlichen Stimme nicht so schnell ändert, besteht eine große Wahrscheinlichkeit, dass die Audiodaten des verlorene Pakets dem vorherigen sehr ähnlich sind und der Unterschied kaum hörbar wird. Alle drei dieser einfachen Methoden für „loss concealment“ sind einfach zu implementieren und besitzen eine geringe Laufzeit, werden dafür vor Allem bei längeren Stille-Perioden sehr schnell hörbar und stören den Gesprächsverlauf. Deshalb gibt es noch eine Vielzahl weiterer Algorithmen, die versuchen, Unterbrechungen möglichst unhörbar zu überdecken, dies jedoch meist auf Kosten erhöhter Laufzeit und Komplexität erkaufen.
Um den Verlust von Paketen aufgrund von „late loss“ gering zu halten und somit eine flüssige und möglichst vollständige Audiokommunikation zu gewährleisten, führt ein Jitter-Buffer eine künstliche Verzögerung zwischen dem Empfangen und dem Abspielen eines Paketes ein. Dadurch, dass ein Paket später abgespielt wird

(der sog. „playout point“ wird nach Hinten verschoben), hat es länger Zeit, beim Empfänger anzukommen, wodurch weniger Pakete wegen „late loss“, also dem Ankommen nach ihrem „playout point“, verworfen werden. Hierfür sollte die Abspielverzögerung so gewählt werden, dass möglichst alle Pakete, die nicht auf dem Netzwerk verloren gehen, den Empfänger vor ihrem Abspielzeitpunkt erreichen. Auf der anderen Seite darf die Abspielverzögerung nicht zu groß werden, sonst leidet die Qualität der Echtzeitkommunikation. Dafür wird in umfangreicheren Implementierungen die Abspielverzögerung meist dynamisch angepasst. Diese sog. „playout point adaption“ wird auf Basis des Anteils an „late loss“ berechnet und in bestimmten Abständen durch einschieben eines zusätzlichen Pakets (z.B. Stille) oder überspringen eines empfangenen Paketes umgesetzt. Der Jitter-Buffer des OHMComm-Frameworks heißt `RTPBuffer` und besitzt in der aktuellen Version eine feste Abspielverzögerung, die bei der Kompilierung des Programms eingestellt werden muss.
Um ein asynchrones Empfangen von RTP-Paketen zu ermöglichen, wird der RTP-Listener (siehe Abschnitt 3.3.8) in einem eigenen Thread umgesetzt. Der RTP-Buffer oder Jitter-Buffer bietet die Schnittstelle zwischen dem Listener-Thread zum Empfangen von RTP-Paketen und der Verarbeitungskette, die die darin enthaltenen Audiodaten weiter behandelt. Dafür muss der Jitter-Buffer thread-safe implementiert sein, also gegen Data Races in den Puffereinträgen und anderen Werten abgesichert sein.
Des Weiteren muss der Jitter-Buffer – wie auch jede andere Komponente in der Verarbeitungskette – performant umgesetzt werden, da jede Verzögerung in einer der Glieder der Verarbeitungskette die gesamte Aufnahme- oder Abspielverzögerung vergrößert, wodurch sich die Gesprächsqualität verschlechtert. Dafür wird der Jitter-Buffer – so wie viele performance-kritische Puffer – als Ringpuffer implementiert. Ein Ringpuffer ist als Array auf dem Speicher angelegt. Indem der Lese- oder Schreibindex am Ende des Arrays umgeschlagen wird (`modulo Buffergröße`), kann der Puffer als endlose Datenstruktur verwendet werden, die jedoch nur die letzten `x` Einträge besitzen kann (wobei `x` die Größe des Puffers ist). Dies stellt für den Jitter-Buffer keine Beeinträchtigung dar, da sowieso alle Pakete, die bereits gelesen worden sind, nicht mehr gebraucht werden und somit ein RTP-Buffer nur eine Hand voll neue Pakete halten muss. Die Vorteile eines Ringpuffers bestehen aus einem konstanten Speicherverbrauch im Gegensatz zu den vielen Allokationen und Deallokationen, die z.B. eine verkettete Liste benötigen würde, beim Hinzufügen und Entfernen von Paketen.

3.3.7 Netzwerkverbindung

Wie auch die meisten anderen Komponenten des OHMComm-Framework (Audio-Bibliothek, -Prozessoren) ist auch die Umsetzung der Netzwerkschnittstelle austauschbar implementiert. Dies wird durch die abstrakte Basisklasse `NetworkWrapper` umgesetzt, die Methodensignaturen zum Senden und Empfangen von Daten sowie zum Schließen der Verbindung bereitstellt. Bei einer Implementierung einer Netzwerkschnittstelle muss dafür gesorgt werden, dass die Senden- und Empfangen-Methoden keine Data-Races erzeugen können, da diese eventuell aus verschiedenen Threads heraus aufgerufen werden. Da RTP in den allermeisten Fällen mit dem Transportprotokoll UDP verwendet wird, gibt es jedoch derzeit nur eine konkrete Implementierung der Netzwerkschnittstelle namens `UDPWrapper`, die RTP-Pakete in UDP-Pakete verpackt und beim Empfangen wieder entpackt. Spätere Versionen des Frameworks enthalten zusätzlich noch eine Implementierung, die TCP verwendet.

3.3.8 RTPListener

Die Socketmethoden zum Empfangen von Paketen sind blockierend [Microsoft, 2000]. Dies hat weitreichende Folgen für die Architektur. Der Ausgangspunkt der Verarbeitung ist die Verarbeitungskette im `AudioHandler`. Wird dort eine blockierende Funktion aufgerufen, so wird die gesamte Weiterverarbeitung blockiert. Wenn z.B. innerhalb eines Audioprozessors die blockierende Socket-Funktion `recvfrom()` aufgerufen wird, und es werden keine Pakete empfangen, dann bricht die gesamte Verarbeitung ab. Dieses Verhalten ist nicht erwünscht. Die Verarbeitung soll unabhängig von den empfangenen Paketen funktionieren. Die Klasse `RTPListener` stellt einen Lösungsansatz für dieses Problem dar. Hierfür wird der gesamte Empfangsprozess in einem eignen Thread ausgelagert. Abbildung 3.7 zeigt den Aufbau des `RTPListeners`.

Die Methode `startUp()` startet den `RTPListener` in einem separaten Thread und `shutdown()` beendet ihn wieder. Der Empfangsprozess besteht aus den Komponenten `RTPBuffer`, `RTPPackageHandler` und dem `NetworkWrapper`. Im Thread des `RTPListeners` wird in einer Endlosschleife die blockierende Funktion `receiveData()` des `NetworkWrappers` aufgerufen. Als Parameter wird der Funktion der Buffer des `RTPPackageHandlers` übergeben. Hierfür wird die Funktion `getWorkBuffer()` verwendet. Wenn ein Paket empfangen wurde, so kann mit Hilfe von `RTPPackageHandler` der RTPHeader ausgelesen. Falls keine weitere Verarbeitung nötig ist, kann das Paket im `RTPBuffer` mit `addPackage()`

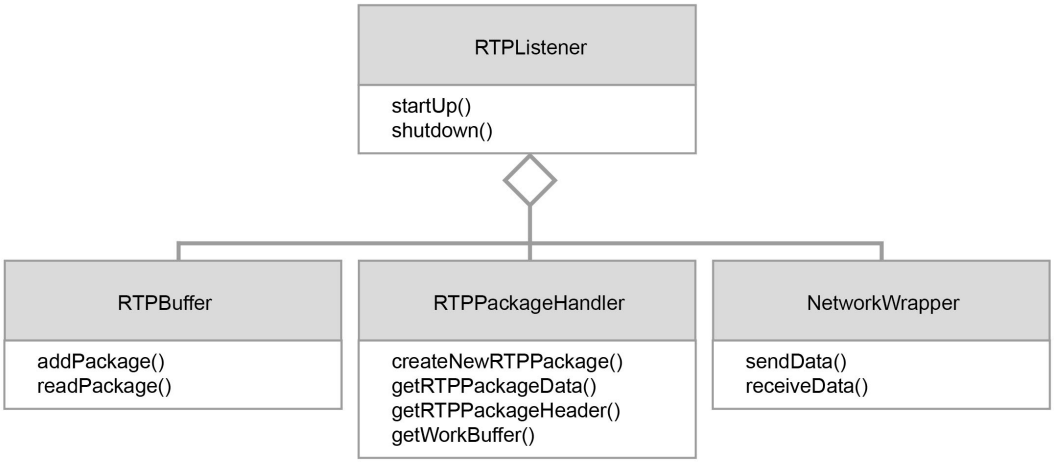


Abbildung 3.7: Der gesamte Empfangsprozess in einem separaten Thread

zwischengespeichert werden. Anschließend kann es mit `readPackage()` wieder ausgelesen werden. Das Auslesen des Paket erfolgt in der Regel vom Main-Thread, welches ohne Probleme möglich, da der `RTPBuffer` thread-safe ist (siehe Abschnitt 3.3.6).

3.4 Konkrete Softwarekomponenten

3.4.1 RtAudio

Als **Audiowrapper** wird RtAudio [Scavone, 2014a] eingesetzt. RtAudio wird entwickelt von *Gary P. Scavone*, Professor an der McGill Uni in Kanada, und wird unter der *MIT-Lizenz* vertrieben [Scavone, 2014c]. RtAudio dient als Programmierschnittstelle, engl. Application Programming Interface(API), zwischen den Betriebssystemabhängigen APIs und OHMComm. RtAudio unterstützt eine Vielzahl von APIs. Aufgeschlüsselt nach Betriebssystem sind diese [Scavone, 2014b]:

Microsoft Windows:

- Audio Stream Input/Output (ASIO)
- DirectSound (DS)
- Windows Audio Session API (WASAPI)

Mac OS X:

- CoreAudio

- Jack Audio Connection Kit (Jack)

GNU/Linux:

- Advanced Linux Sound Architecture (ALSA)
- Open Sound System (OSS)
- PulseAudio
- Jack Audio Connection Kit (Jack)

Ein alternativer plattformunabhängiger Audiowrapper wäre *PortAudio* [Bencina, 2014], welcher jedoch in C geschrieben ist. Da OHMComm und RtAudio in C++ geschrieben sind, ist dies die bessere Kombination.

3.4.2 Opus

Als **Audiocodec** wird Opus [Xiph.Org, 2016a] eingesetzt. Opus ist ein Codec zur verlustbehafteten Kompression von Audiodaten. Opus ist speziell für *Echtzeitanwendungen* konzipiert und deshalb bestens geeignet für das OHMComm-Framework. Er wird unter einer *OpenSource-Lizenz* [Xiph.Org, 2016b] als royalty-free Codec, also ein Codec ohne Lizenzkosten, vertrieben. Opus ist ein *Hybridcodec*, welcher eine Transformationsschicht **CELT** (Entwickelt von der Xiph.Org Foundation) [Xiph.Org, 2011] und ein Linear Predictive Coding(LPC) namens **SILK** (Ursprünglich Entwickelt von Skype Technologies) [Skype, 2012] zur Audiokompression einsetzt.

Viele bekannte Anwendungen setzten ebenfalls auf Opus als Audiocodec, darunter *TeamSpeak* [TeamSpeak, 2013], die *Webbrowser* Firefox [Mozilla, 2016] und Chrome [Google, 2011] – innerhalb ihrer WebRTC Implementierungen, da dort Opus vom Standard vorausgesetzt wird [IETF, 2015] – und *Android 5* (Lollipop), welches eine native Unterstützung von Opus bietet [Open-Handset-Alliance, 2014].

Weitere Feature von Opus sind [Xiph.Org, 2015]:

- Unterstütze Sampling Raten: 8000, 12000, 16000, 24000, und 48000Hz
- 3 Application Modi:
  - OPUS_APPLICATION_VOIP: Optimiert auf Sprachverständlichkeit
  - OPUS_APPLICATION_AUDIO: Optimiert auf Klanggenuss, für Musik
  - OPUS_APPLICATION_RESTRICTED_LOWDELAY: Optimiert auf kleinstmögliche Latenz

- Audioframegrößen von 2.5 ms bis 60 ms
- Mono und Stereo Support

Alternative Codecs wären *AAC-LD* oder *AAC-ELD*, welche aber Lizenzkostenpflichtig [Fraunhofer-IIS, 2016] sind und deshalb nicht für OHMComm in Frage kommen.

3.5 Statistiken

Ein weiteres Feature des OHMComm-Frameworks ist das Sammeln und Anzeigen von statistischen Daten (siehe Anforderung g). Die verschiedenen Statistiken wurden eingebaut, um die Performance einzelner Abschnitte der Verarbeitungskette zu messen, sowie die Richtigkeit der Handhabung von RTP-Paketen und frei gewählter Werte (wie der Buffergröße und Abspiellatenz) zu beweisen. Im Laufe des Programms werden eine Vielzahl an verschiedenen statistischen Daten gesammelt. Darunter fallen:

Audio-Daten: Bei jedem Durchlauf der Verarbeitungskette werden Zähler für die Gesamtzahl der aufgenommenen sowie abgespielten Samples und Daten-Bytes erhöht.

RTP-Daten: Ebenso werden die Anzahl an versendeten und empfangenen RTP-Paketen, sowie die Größe deren Nutzdaten und der RTP-Header gezählt. Für das bestimmen der Quality of Service werden die Anzahl der verlorenen Pakete gezählt, sowie der maximale Füllstand des Jitter-Buffers gespeichert.

Prozessor-Daten: Wenn das messen der Prozessorlaufzeiten aktiviert ist, wird für jeden Audioprozessor in der Verarbeitungskette die Gesamtlaufzeit der beiden Methoden zum Bearbeiten der aufgenommenen oder abgespielten Daten gemessen.

Aus diesen gemessenen Werten werden beim Beenden des Frameworks verschiedene relevante Statistiken berechnet und ausgegeben. Die Ausgabe der Statistiken erfolgt immer über den Standardoutput (meist eine Konsole) sowie (falls vorher so konfiguriert) in eine Datei. Angezeigt werden die folgenden statistischen Werte:

- Die Gesamtzahl der aufgenommen und abgespielten Audiodaten (in Bytes) sowie Samples. Ebenso werden diese Werte durch die Gesamtlaufzeit der Kommunikation geteilt, um den Durchsatz der Soundkarte zu berechnen. An den Ergebnissen (vor allem an der Input- und Output-Samplerate) lässt sich erkennen, dass die Soundkarte sich nicht ganz an die vorher konfigurierte Samplerate hält, sondern leicht davon abweicht.

- Die Gesamtzahl der gesendeten und empfangenen Daten (in Bytes) sowie Pakete. Wie auch im vorherigen Punkt sind hier die Durchsatz-Raten von größerer Bedeutung als die Gesamtzahlen. So wird hier der Datendurchsatz für die gesendeten und empfangene Pakete ausgerechnet, also die wirklich benötigte Netzwerkbandbreite in beide Richtungen. Ebenso werden die Bandbreiten der reinen Audiodaten und der prozentuale Overhead der RTP-Header ausgegeben. Der Header-Overhead ist abhängig von der Größe der Daten in einem RTP-Paket und liegt meist unter 5%.

- Die Anzahl der verlorenen Pakete (also nicht empfangene Sequenznummern, siehe Abschnitt 3.3.5) sowie der maximale Füllstand der Jitter-Buffers. Anhand dieser Werte kann die Übertragungsqualität sowie die maximale Abspielverzögerung (der maximale „playout point“) abgelesen werden. Für beide Werte gilt, je kleiner der Wert, desto besser ist die Qualität der Übertragung.

- Aus der Gesamtmenge an aufgenommenen/abgespielten Audiodaten und der Gesamtmenge an gesendeten/empfangen Audiodaten lässt sich die Kommpressions-/Dekompressionsrate berechnen, also wie stark ein verwendeter Audiocodec die Größe der Audiodaten komprimiert. Bei einer Kommunikation ohne zugeschalteten Audiocodec wird exakt die gleiche Anzahl an Bytes gesendet, die aufgenommen wird oder auch abgespielt, die empfangen wird, wodurch eine Kompression von 0% angezeigt wird. Eine Verwendung des Opus-Codecs resultiert z.B. in einer Kompression von ca. 96%, also werden nur 4% der aufgenommen Audio-Bytes über das Netzwerk versendet und auch aus den empfangenen 4% beim Entpacken wieder die volle Anzahl an Audiodaten wieder hergestellt. Daran kann man sehr gut die Bedeutung eines Audiocodecs für das Minimieren der benötigten Bandbreite erkennen.

- Die Laufzeiten der verschiedenen Audioprozessoren. Wenn das Messen der Ausführungszeiten der Verarbeitungskette aktiviert wird, wird für jeden verwendeten Audioprozessor die Gesamtausführungszeiten der Methoden zum Bearbeiten des Audio-Inputs und -Outputs angezeigt. Auch hier wird zusätzlich der interessantere Wert der Ausführungszeit pro Aufruf ausgegeben, an dem die sog. algorithmische Verzögerung eines einzelnen Prozessors und der ganzen Verarbeitungskette abgelesen werden kann. Wenn die algorithmische Verzögerung der gesamten Prozessorkette zu groß wird, bekommt die Soundkarte nicht rechtzeitig neue Audiodaten und es wird ein hörbares Knacken ausgegeben.

3.6 Dokumentation

Für die Dokumentation wird das Tool Doxygen verwendet. Es unterstützt eine Vielzahl an Programmiersprachen und ist plattformunabhängig. Die Dokumentation des Quellcodes erfolgt im Quellcode selbst. Hier zu werden Kommentare mit besonderer Syntax verwendet, aus denen später die Dokumentation generiert werden kann. Dokumentationsformate sind HTML, RTF, PostScript, PDF oder Unix-Manpages. Es gibt unterschiedliche Stile für Dokumentationskommentare. OHMComm verwendet den QT-Style, da dieser häufig für C++ Projekte eingesetzt wird. Listing 3.3 zeigt wie eine solche Dokumentation im Quellcode aussehen kann. Kommentare zum Dokumentieren werden mit `/*!` eingeleitet. Die Parameter werden mit param und die Rückgabewerte mit return beschrieben. Doxygen erwartet, als Konsolenanwendung, einen Parameter mit den Einstellungen für die Dokumentation zum Generierien. Der Aufruf erfolgt mit `doxygen <config-file>`.

```
/*! Beschreibung der Methode
 * \param a Parameterbeschreibung
 * \param b Parameterbeschreibung
 * \return Was wird zuckgegeben
 */
int testMe(int a, char *b);
```

Listing 3.3: Code-Dokumentation im QT-Style

Kapitel 4

Implementierung

In diesem Kapitel wird auf Implementierungsdetails des Entwurfs eingegangen. Hierbei ist zu beachten, dass aus Platzgründen der vorliegende Code eventuell in gekürzter oder geänderter Form dargestellt wird und unvollständig ist. Dennoch verdeutlichen die Codebeispiele die wesentlichsten Details der Implementierung und sind gemäß des Entwurfs umgesetzt.

4.1 Verarbeitungskette

Die Verarbeitungskette ist ein Teil des `AudioHandlers` und einer der wichtigen Komponenten des Frameworks. Sie ist die Schnittstelle, mit der anderen Klassen an der Audioverarbeitung teilnehmen können. Im folgenden Kapitel wird erörtert, wie diese implementiert wurde.

4.1.1 Interface für Audioverarbeitungsklassen

Alle Audioverarbeitungsklassen müssen die abstrakte Klasse `AudioProcessor` implementieren, siehe Listening 4.1.

```
1 class AudioProcessor {
2 public:
3  AudioProcessor(const std::string name);
4  virtual unsigned int processInputData(void *inputBuffer, int
      inputBufferByteSize) = 0;
5  virtual unsigned int processOutputData(void *outputBuffer,  int
      outputBufferByteSize) = 0;
6 }
```

Listing 4.1: Interface des AudioProcessors

Der Konstruktor in Zeile 4 erwartet als Parameter den Namen des Audioprozessors, damit dieser später eindeutig identifiziert werden kann. Die beiden Verarbeitungsmethoden in Zeile 5 und 6 sind virtuell, daher müssen alle Unterklassen diese implementieren. Diese Methoden stellen die Schnittstelle zur Audioverarbeitung dar und haben als Parameter jeweils den Buffer sowie die Buffergröße.

4.1.2 An- und Abmeldeprozess im AudioHandler

Listening 4.2 zeigt die Schnittstelle zum An- und Abmelden von Audioprozessoren im `AudioHandlers` an. Die Methode `addProcessor()` fügt einen Audioprozessor in die Audioverarbeitungsliste ein, falls er noch nicht vorhanden ist, und `removeProcessor()` ist die entsprechende Löschmethode. Als Liste wird hier der Standardcontainer `std::vector` benutzt. Bei diesem werden neue Elemente am Ende der Liste eingefügt.

```
1  bool AudioHandler::addProcessor(AudioProcessor *audioProcessor)
2  {
3   if (hasAudioProcessor(audioProcessor) == false) {
4    audioProcessors.push_back(std::unique_ptr<AudioProcessor>(
       audioProcessor));
5    return true;
6   }
7   return false;
8  }
9
10 bool AudioHandler::removeAudioProcessor(AudioProcessor *audioProcessor) {
11  for (size_t i = 0; i < audioProcessors.size(); i++)
12  {
13   if ((audioProcessors.at(i))->getName() == audioProcessor->getName()) {
14    audioProcessors.erase(audioProcessors.begin() + i);
15    return true;
16   }
17  }
18  return false;
19 }
```

Listing 4.2: An- und Abmeldeprozess von Audioprozessoren im AudioHandler

4.1.3 Verarbeitungsprozess

Listening 4.3 zeigt den Verarbeitungsprozess. Die Methode `processAudioInput()` wird vom AudioHandler aufgerufen, falls Daten vom Mikrofon zur Verarbeitung anliegen. Dort wird von jedem angemeldeten Audioprozessor die entsprechende `processInputData()` -Methode aufgerufen. Als Parameter werden die Daten

des Mikrofons und deren Datengröße übergeben. Die Verarbeitungsreihenfolge entspricht der Anmeldereihenfolge. Die Verarbeitung im Buffer erfolgt in-place, daher auf den übergebenen Buffer. Falls ein Audioprozessor zu Beginn die Daten manipuliert, so sind diese Daten für alle nachfolgenden Prozessoren ebenfalls geändert. Im Umkehrschluss heißt dies, um wieder an die Ursprungsdaten zu gelangen, müssen die ausgeführten Änderungen in umgekehrte Reihenfolge rückgängig gemacht zu werden. `processAudioOutput()` wird aufgerufen, wenn die Soundkarte bereit ist Audiodaten abzuspielen. Hierzu werden von allen Audioprozessoren die `processOutputData()` - Methoden aufgerufen, jedoch mit umgekehrter Aufrufreihenfolge, siehe Zeile 13. Dadurch wird der Ursprungszustand der Audiodaten hergestellt und sind abspielbar.

```
1  void AudioHandler::processAudioInput(void *inputBuffer,  int
       inputBufferByteSize)
2  {
3   unsigned int bufferSize = inputBufferByteSize;
4   for (unsigned int i = 0; i < audioProcessors.size(); i++)
5   {
6    bufferSize = audioProcessors.at(i)->processInputData(inputBuffer,
       bufferSize);
7   }
8  }
9
10 void AudioHandler::processAudioOutput(void *outputBuffer, int
       outputBufferByteSize)
11 {
12  unsigned int bufferSize = outputBufferByteSize;
13  for (unsigned int i = audioProcessors.size(); i > 0; i--)
14  {
15   bufferSize = audioProcessors.at(i-1)->processOutputData(outputBuffer,
       bufferSize);
16  }
17 }
18 }
```

Listing 4.3: Verarbeitungsprozess des AudioHandlers

4.2 Factory-Klassen

Die beiden Factory-Klassen `AudioHandlerFactory` und `AudioProcessorFactory` sind vom Aufbau identisch, deshalb wird im folgenden nur die `AudioHandlerFactory` erklärt. `AudioProcessorFactory` verhält sich analog dazu. Die Factory-Klassen haben die Aufgabe die Instantiierung in Methoden auszulagern, damit das Programmieren auf Schnittstellen gewährleistet wird. Methoden die Instanti-

ierung übernehmen werden auch Fabrikmethoden genannt [Goll and Dausmann, 2013][S.243]. Listening 4.4 zeigt die Fabrikmethode der `AudioHandlerFactory`. Als Parameter wird der Name der zu erzeugende Klasse entgegen genommen. Der Rückgabewert der Funktion entspricht den abstrakten und gemeinsamen Basistypen.

```
1 std::unique_ptr<AudioHandler> AudioHandlerFactory::getAudioHandler(std::
      string name) {
2  if (name == RTAUDIO_WRAPPER)
3  {
4   std::unique_ptr<RtAudioWrapper> rtaudiowrapper(new RtAudioWrapper);
5   return std::move(rtaudiowrapper);
6  }
7  throw std::invalid_argument("No AudioHandler for this name!");
8 }
```

Listing 4.4: Fabrikmethode der AudioHandlerFactory

4.3 RTPListener

`RTPListener` ermöglicht das asynchrone Empfangen von Paketen durch das Erstellen eines eignen Threads. Die wichtigsten Methoden der Klasse sind im Listening 4.5 dargestellt. Die `startUp()`-Methode startet den Thread und `shutdown()` beendet ihn wieder. Sobald der Thread startet wird `runThread()` aufgerufen. Tatsächlich ist diese Methode komplexer, wurde jedoch für das Beispiel auf das wesentliche gekürzt. In der Schleife wird die blockierende `receiveData()`-Funktion aufgerufen. Übertragene Packete werden im Jitter-Buffer zwischengespeichert, siehe Zeile 14. Die empfangenen Pakete können anschließend aus dem Jitter-Buffer von anderen Threads ausgelesen werden. Der `RTPListener` wartet anschließend wieder auf neue ankommende Pakete.

```
1  void RTPListener::startUp()
2  {
3      threadRunning = true;
4      receiveThread = std::thread(&RTPListener::runThread, this);
5  }
6
7  void RTPListener::shutdown(){
8   threadRunning = false;
9  }
10
11 void RTPListener::runThread() {
12  while(threadRunning) {
13   int receivedSize = this->wrapper->receiveData(rtpHandler.getWorkBuffer
       (), rtpHandler.getMaximumPackageSize());
```

```
14    auto result = buffer->addPackage(rtpHandler, receivedSize -
        RTP_HEADER_MIN_SIZE);
15   }
16  }
```

Listing 4.5: Die wichtigsten Methoden des RTPListeners

4.4 RtAudio

Der **RtAudioWrapper** bietet drei Modi zur Audiokommunikation, diese sind *RecordingMode*, *PlaybackMode* und *DuplexMode*. Bei den ersten beiden Modi wird entweder nur der Input- oder der Output-Stream übergeben um nur Sound aufzunehmen oder abzuspielen. Dagegen benutzt der letzte Modus(DuplexMode) den input und output Stream um ein simultanes aufnehmen und abspielen von Audio Daten zu ermöglichen.

Die RtAudio-API wird mittels eines Objekts, dessen Namen auf rtaudio definiert wurde gesteuert.

```
1 RtAudio rtaudio;
2 RtAudio::StreamParameters input, output;
```

Listing 4.6: RtAudio Objekt und StreamParameter

Um den *Duplex Modus* zu starten wird zuerst per Flag (`flagPrepare`) geprüft ob die `prepare()`-Methode von RtAudioWrapper bereits ausgeführt worden ist – welche überprüft ob die Output und Input Stream Parameter konfiguriert sind – und ob alle AudioProzessoren mittels der `prepare()`-Methode des AudioHandler ihre Unterstützten Parameter gemeldet haben und sich auf einen gleichen Parametersatz einigen konnten. Wenn dies nicht der Fall ist, wird ein Fehler ausgegeben, mit dem Hinweis, dass die AudioProzessoren zuerst konfiguriert werden müssen, bevor der RtAudio Stream gestartet werden kann.

Dem `rtaudio`-Objekt werden per `openStream()`-Methode die benötigten Parameter mitgeteilt, welche folgenden Inhalt haben:

output Der Output Stream Parameter, welcher die `deviceID`, also die ID des Hardware Geräts, welches die Audiosignale ausgeben soll und die Anzahl der zu benutzenden *Channels* enthält. Diese sind entweder per `setDefaultAudioConfig()`-Methode des RtAudioWrapper oder per *FileConfiguration*, *InteractiveConfiguration* oder *ParameterConfiguration* vorher festgelegt worden.

input Der Input Parameter welcher die `deviceID` des Audiogeräts, welches die Audiosignale aufnimmt, und die Anzahl der aufzunehmenden *Channels* ent-

hält. Dieser ist ebenfalls bereits mittels einer der verschiedenen Konfigurationsmodi konfiguriert worden.

audioConfiguration.audioFormatFlag : Das Audioformat-Flag, welches das *Audioformat* definiert(z.B. SINT16 oder FLOAT32), welches durch den AudioHandler durch Abfragen der AudioProzessoren ausgehandelt wurde.

audioConfiguration.sampleRate Die *Sample Rate*, die vom AudioHandler durch Abfragen der AudioProzessoren ausgehandelt wurde.

audioConfiguration.framesPerPackage Die Anzahl der *Frames* per Package, also die Größe des Audiobuffers. Diese Größe wurde auch vom AudioHandler durch Abfragen der AudioProzessoren ausgehandelt

RtAudioWrapper::callbackHelper Die *Callback-Methode*, welche von der RtAudio-API aufgerufen wird, wenn Audiodaten vorhanden sind.

this Eine Referenz auf das eigene *Callback-Objekt*, um innerhalb der Callback-Methode auf benötigte Informationen zuzugreifen.

Anschließend wird die RtAudio-API mittels `startStream()` gestartet.

```
1 void RtAudioWrapper::startDuplexMode()
2 {
3  if (this->flagPrepared)
4  {
5   this->rtaudio.openStream(&output, &input, audioConfiguration.
      audioFormatFlag,   audioConfiguration.sampleRate, &audioConfiguration.
      framesPerPackage, &RtAudioWrapper::callbackHelper, this);
6   this->rtaudio.startStream();
7  }
8 }
```

Listing 4.7: Start des Duplex Mode im RtAudioWrapper

Der `callbackHelper()` erstellt daraufhin ein `RtAudioWrapper`-Objekt und ruft dessen `callback()`-Methode auf. Diese Aufteilung wurde gewählt da hierdurch besser ersichtlich ist wo es sich um *Eingabedaten*(von der RtAudio-API `callbackHelper()`-Methode) und wo um *Ausgabedaten*(zu den AudioProzessoren `callback()`-Methode) handelt.

```
1 auto RtAudioWrapper::callbackHelper(void *outputBuffer, void *inputBuffer,
     unsigned int nBufferFrames, double streamTime, RtAudioStreamStatus
    status, void *rtAudioWrapperObject) -> int
2 {
3  RtAudioWrapper *rtAudioWrapper = static_cast <RtAudioWrapper*> (
     rtAudioWrapperObject);
```

```
4   return rtAudioWrapper->callback(outputBuffer, inputBuffer, nBufferFrames,
        streamTime, status, nullptr);
5  }
```

Listing 4.8: `callbackHelper()`-Methode im RtAudioWrapper

Die `callback()`-Methode des RtAudioWrapper ruft die Methoden `processAudioInput()` und `processAudioOutput()` auf welche die einzelnen Prozessoren durchläuft und folgende Parameter enthält :

inputBuffer Pointer auf den Speicherbereich der die input Audio Daten enthält.

outputBuffer Pointer auf den Speicherbereich der die output Audio Daten enthält.

inputBufferByteSize Größe der Daten die im inputBuffer vorhanden sind.

outputBufferByteSize Größe der Daten die im outputBuffer vorhanden sind.

streamData Struct welcher zusätzliche Informationen zu den Daten im Audio Buffer enthält, wie die Anzahl der Frames im Buffer(`nBufferFrames`), die Aktuelle Zeit des Streams(`streamTime`) und die Maximale Anzahl der Daten welche in den Buffer geschrieben werden könne(`maxBufferSize`).

```
1  auto RtAudioWrapper::callback(void *outputBuffer, void *inputBuffer,
       unsigned int nBufferFrames, double streamTime, RtAudioStreamStatus
       status, void *rtAudioWrapperObject) -> int
2  {
3   (...)
4   this->processAudioInput(inputBuffer, inputBufferByteSize, streamData);
5   (...)
6   this->processAudioOutput(outputBuffer, outputBufferByteSize, streamData);

7  }
```

Listing 4.9: callback Methode im RtAudioWrapper

4.5 Opus

Der **Opus-Codec**, welcher zum *Encoden* und *Decoden* der Audiodaten benutzt wird wurde als *Prozessor* in OHMComm eingebunden.

Der *Konstruktor* des Opus-Prozessors erstellt bei der Instanziierung jeweils ein Encoder- und Decoder-Objekt. Außerdem wird ihm ein Name gegeben und der Opus-Application-Type (siehe 3.4.2) wird gewählt.

```
1 ProcessorOpus::ProcessorOpus(const std::string name, int opusApplication)
      : AudioProcessor(name), OpusEncoderObject(nullptr), OpusDecoderObject
    (nullptr)
2 {
3     this->OpusApplication = opusApplication;
4 }
```

Listing 4.10: Instanziierung des Opus Prozessors

Anschließend werden die Encoder- und Decoder-Objekte in der `configure()`-Methode konfiguriert. Dies geschieht mit dem Aufruf der `opus_encoder_create()` bzw. `opus_decoder_create()`-Methoden. Die Parameter hierbei sind:

audioConfig.sampleRate Die *Sample Rate* der Audio Daten. Welche einen der von Opus unterstützen Werte haben muss siehe 3.4.2.

audioConfig.inputDeviceChannels Die Anzahl der input *Channels*.

audioConfig.outputDeviceChannels Die Anzahl der output *Channels*.

OpusApplication Der *Opus-Application-Type* siehe 3.4.2.

ErrorCode Eine Variable welche im Fehlerfall den *Error Code* zur Fehlerbehandlung enthält.

```
1 bool ProcessorOpus::configure(const AudioConfiguration& audioConfig,
    const std::shared_ptr<ConfigurationMode> configMode)
2 {
3  (...)
4  OpusEncoderObject = opus_encoder_create(audioConfig.sampleRate,
     audioConfig.inputDeviceChannels, OpusApplication, &ErrorCode);
5  OpusDecoderObject = opus_decoder_create(audioConfig.sampleRate,
     audioConfig.outputDeviceChannels, &ErrorCode);
6  (...)
7 }
```

Listing 4.11: Konfiguration der Opus Encoder und Decoder-Objekte

Das *Encodieren* der Audiodaten geschieht in der `processInputData()`-Methode des Opus-Prozessors. Zu Beginn wird die Variable `lengthEncodedPacketInBytes` deklariert in der die Größe der Audiodaten nach ihrer Encodierung gespeichert wird. Anschließend wir geprüft ob Daten im 16-bit Signed-Integer-Format oder im 32-bit Float-Format vorliegen. Abhängig davon wir die entsprechende `opus_encode()`-Methode aufgerufen, welche folgende Parameter benötigt:

OpusEncoderObject Das bei der Instanziierung erzeugte und in der `configure()`-Methode konfigurierte Opus *Encoder Objekt*.

inputBuffer Der ins 16-bit Signed-Integer oder 32-bit Float Format gecastete *InputBuffer* Pointer welcher die Input Audio Daten der Soundkarte enthält.

userData->nBufferFrames Die Anzahl der *Samples*(per Channel) welche sich im InputBuffer befinden. Da Opus Audioframegrößen von 2.5ms bis 60ms unterstützt (siehe 3.4.2), muss die Anzahl der Samples auch in diesem Bereich liegen. Bei 48kHz Sample Rate und einer Audioframegröße von 20ms also 960 Samples pro Channel.

inputBuffer Char-Pointer der Anzeigt wohin die Encodierten Audio Daten geschrieben werden sollen. Da wir auf dem *InputBuffer* Arbeiten und keinen extra Speicherplatz für Encodierte Daten haben, ist dies der selbe Speicherbereich aus dem wir die Daten gelesen haben.

userData->maxBufferSize Maximal Größe des Speicherbereichs wohin die Encodierten Daten geschrieben werden.

Als Rückgabewert geben wir die neue Größe des `inputBuffers` weiter.

```
1  unsigned int ProcessorOpus::processInputData(void *inputBuffer, const
       unsigned int inputBufferByteSize, StreamData *userData)
2  {
3      unsigned int lengthEncodedPacketInBytes = 0;
4      if (rtaudioFormat == AudioConfiguration::AUDIO_FORMAT_SINT16)
5      {
6          lengthEncodedPacketInBytes = opus_encode(OpusEncoderObject, (
       opus_int16 *)inputBuffer, userData->nBufferFrames, (unsigned char *)
       inputBuffer, userData->maxBufferSize);
7          return lengthEncodedPacketInBytes;
8      }
9      else if (rtaudioFormat == AudioConfiguration::AUDIO_FORMAT_FLOAT32)
10     {
11         lengthEncodedPacketInBytes = opus_encode_float(OpusEncoderObject,
        (const float *)inputBuffer, userData->nBufferFrames, (unsigned char
       *)inputBuffer, userData->maxBufferSize);
12         return lengthEncodedPacketInBytes;
13     }
14 }
```

Listing 4.12: Encodieren von Audio Daten mittels Opus

Das *Decodieren* geschieht auf ähnliche Weise wie das Encodieren der Audio Daten innerhalb der `processOutputData()`-Methode. Zuerst wird eine Variable angelegt, in der die Anzahl der decodierten Samples gespeichert wird(`numberOfDecoded-Samples`), welche von der `opus_decode()`-Methode als Rückgabewert zurückgeliefert wird. Es wird wieder geprüft, ob die Daten im 16-bit Signed-Integer-

Format oder im 32-bit Float-Format decodiert werden sollen. Anschließend wird die `opus_decode()`-Methode mit folgenden Parametern aufgerufen:

OpusDecoderObject Das bei der Instanziierung erzeugte und in der `configure()`-Methode konfigurierte Opus *Decoder Objekt*.

outputBuffer Char-Pointer auf die Encodierten Audio Daten.

outputBufferByteSize Größe des OutputBuffers.

outputBuffer Pointer wohin die Decodierten Audio Daten geschrieben werden sollen entweder im Signed Integer 16 oder Float 32 Format.

userData->maxBufferSize Maximal Anzahl der *Samples* welche in den Speicherbereich der Decodierten Daten geschrieben werden können.

0 Möglicher Platz für ein *Flag* welches Anzeigt ob in-band forward error correction Daten vorhanden sind, was nicht der Fall ist.

Da wir als *Rückgabewert* von `opus_decode()` die Anzahl der decodierten Samples bekommen, die Prozessoren aber als Rückgabewert die Größe des OutputBuffers zurückgeben sollen, muss diese noch Mittels der Anzahl der decodierten Samples multipliziert mit der Größe des gewünschten Formates multipliziert mit der Anzahl der Channels `(numberOfDecodedSamples * sizeof(opus_int16) * outputDeviceChannels)` berechnet werden.

```
1  unsigned int ProcessorOpus::processOutputData(void *outputBuffer, const
       unsigned int outputBufferByteSize, StreamData *userData)
2  {
3      unsigned int numberOfDecodedSamples = 0;
4      if (rtaudioFormat == AudioConfiguration::AUDIO_FORMAT_SINT16)
5      {
6          numberOfDecodedSamples = opus_decode(OpusDecoderObject, (unsigned
        char *)outputBuffer, outputBufferByteSize, (opus_int16 *)outputBuffer
       , userData->maxBufferSize, 0);
7          userData->nBufferFrames = numberOfDecodedSamples;
8          const unsigned int outputBufferInBytes = (numberOfDecodedSamples
       * sizeof(opus_int16) * outputDeviceChannels);
9          return outputBufferInBytes;
10     }
11     else if (rtaudioFormat == AudioConfiguration::AUDIO_FORMAT_FLOAT32)
12     {
13         numberOfDecodedSamples = opus_decode_float(OpusDecoderObject, (
       const unsigned char *)outputBuffer, outputBufferByteSize, (float *)
       outputBuffer, userData->maxBufferSize, 0);
14         userData->nBufferFrames = numberOfDecodedSamples;
```

```
15          const unsigned int outputBufferInBytes = (numberOfDecodedSamples
        * sizeof(float) * outputDeviceChannels);
16          return outputBufferInBytes;
17      }
18  }
```

Listing 4.13: Decodieren von Audio Daten mittels Opus

4.6 Passive Konfiguration

Die passive Konfiguration aus Abschnitt 3.3.1 wird mithilfe von **Application-defined** RTCP-Paketen aus Abschnitt 3.3.5 umgesetzt. Hierfür wird für die Konfiguration einer Sitzung der Klient **A** mit der Option „Konfigurationsanfrage aktivieren“ (oder dem Parameter `-wait-for-passive`) gestartet, um eine Anfrage einer passiven Konfiguration zu ermöglichen. Klient **B** wird mit dem Parameter `-passive` ausgeführt. Klient **A** konfiguriert das OHMComm-Framework, startet jedoch noch keine Kommunikation, sondern nur den RTCP-Thread und wartet dort auf eine Anfrage für eine passive Konfiguration. Klient **B** sendet eine RTCP Application-defined Nachricht an **A**, die die Anfrage einer passiven Konfiguration darstellt. Daraufhin antwortet **A** mit einem weiteren RTCP-Paket, das die Werte der passiven Konfiguration (Abtastrate, Audioformat, Audiocodecs, Anzahl der Kanäle sowie die Puffergröße) beinhaltet. Dieses Paket wird von **B** empfangen und die Konfiguration ausgelesen. Daraufhin können beide Seiten das Übertragen von Audiodaten beginnen. Durch die Übertragung aller relevanten Informationen (Einstellungen der Audiobibliothek sowie die verwendeten Audiocodecs) wird garantiert, dass die gesendeten Daten von der anderen Instanz auch verstanden und verwendet werden können. Um die passive Konfiguration zu starten, wird für den Klienten **B** nur die IP-Adresse und der Port des Klienten **A** benötigt, wohingegen **A** komplett wie bereits beschrieben konfiguriert werden kann. Der Ablauf einer passiven Konfiguration ist schematisch in Listing 4.1 dargestellt:

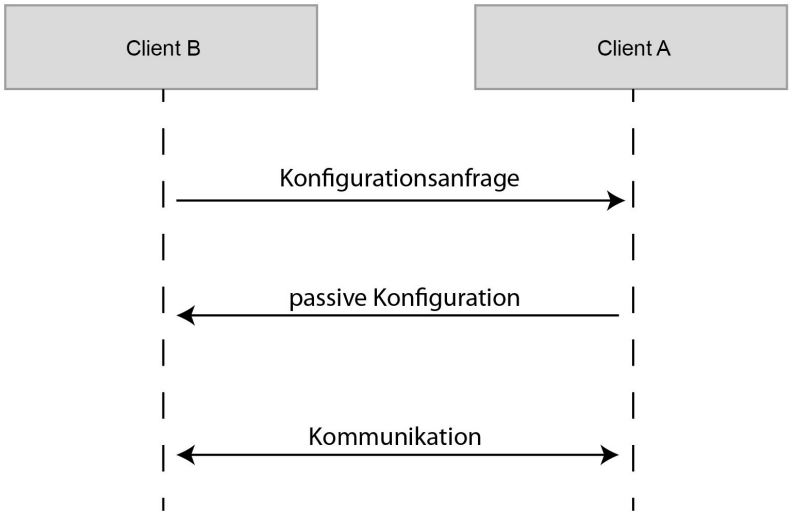


Abbildung 4.1: Ablauf der passiven Konfiguration

Kapitel 5

Prototypische Voice-over-IP Konsolenanwendungen

5.1 Ziel der Anwendung

Um die *Features* des OHMComm-Framework zu zeigen, wurde ein Prototypisches Voice-over-IP Programm als Konsolenanwendung realisiert. Ziel der Konsolenanwendung ist es, die Integration des Frameworks in ein Programm zu verdeutlichen. Außerdem wurden mit der Anwendung in rechnerübergreifenden Tests der *Verbindungsaufbau*, *Verbindungsabbau* und die *Verbindungsqualität* überprüft, das *Zusammenspiel* der einzelnen Komponenten des OHMComm-Frameworks validiert und die *Qualität* der Audioübertragung untersucht.

5.2 Verwendung der Schnittstelle des Frameworks

Das OHMComm-Framework bietet eine sehr übersichtliche Schnittstelle, die verwendet werden muss, um das Framework in eine Anwendung einzubinden. Diese Schnittstelle ist eine Instanz der Klasse `OHMComm`, über die das gesamte Framework gesteuert werden kann. Um diese Schnittstelle nun verwenden zu können, muss ein `OHMComm`-Objekt erstellt werden. Dafür muss vorher eine Konfiguration als Instanz einer der `ConfigurationMode`-Kindklassen erstellt werden. Diese wird dem Konstruktor `OHMComm(ConfigurationMode*)` übergeben und das Framework kann verwendet werden. Daraufhin wird mit der Methode `isConfigurationDone (bool)` überprüft, ob die Konfiguration abgeschlossen ist, die über den Flag-Parameter gestartet werden kann, falls dies nicht so ist. Die restlichen Funktionalitäten werden danach automatisch, je nach übergebener Konfiguration, aktiviert

und eingestellt. Damit ist das Framework vollständig eingestellt und die Kommunikation kann mit einem Methodenaufruf auf `startAudioThreads()` gestartet werden. Dieser Aufruf ist nicht blockierend, so dass der aufrufende Thread weitere Aktivitäten ausführen kann. Solang die Kommunikation läuft, gibt die Methode `isRunning()` `true` zurück. Beendet wird das Framework über die Methode `stopAudioThreads()`. Einen Beispielcode über die Verwendung des Frameworks liefert die prototypische Anwendung in der Datei `OHMCommStandalone.cpp`.

5.3 Steuerung

Die `main()`-Funktion aus `OHMCommStandalone.cpp` implementiert vier der fünf verfügbaren Konfigurationsmodi (siehe Abschnitt 3.3.1). Die prototypische Konsolenanwendung kann also auf vier verschiedene Arten konfiguriert werden: Wird das Programm ohne Kommandozeilenargumenten gestartet, wird die interaktive Konfiguration verwendet, bei der die benötigten Einstellungen nacheinander auf der Konsole mit möglichen Werten ausgegeben werden und der Benutzer durch Texteingabe die Einstellungen setzen muss. Wird das Programm mit einem einzelnen Dateipfad als Parameter gestartet, werden die Einstellungen aus der übergebenen Datei geladen. Wenn das Kommandozeilenargument `-P` oder `-passive` gesetzt ist, wird die zusätzlich als Argument mit übergebene Netzwerkkonfiguration dazu verwendet, um eine Anfrage für eine passive Konfiguration an diese Adresse zu senden (siehe Abschnitt 4.6). Der letzte implementierte Konfigurationsmodus ist die Parameterkonfiguration, bei der alle Einstellungen als Kommandozeilenargumente an den Programmaufruf angehängt werden, wobei mit `-h` oder `-help` eine Übersicht über alle möglichen Einstellungen ausgegeben werden kann.

5.4 Anwendungen ordnungsgemäß beenden

Ein größeres Problem bei der Umsetzung der prototypischen Konsolenanwendung ist es gewesen, die Anwendung ordnungsgemäß zu beenden. Besonders problematisch wird es, da die Anwendung aus verschiedenen Threads besteht, die alle beendet werden müssen, sowie die Kommunikation selber aus verschiedenen Threads heraus beendet werden kann. So müssen folgende Threads beendet werden: Der Thread der Verarbeitungskette, der von der Audioschnittstelle heraus gestartet wird, der RTPListener-Thread und der RTCP-Thread, die beide vom `OHMComm`-Objekt erzeugt werden. Ebenso muss der `NetworkWrapper` geschlossen werden, da dieser auch verbindungsorientiert (z.B. mit TCP) implementiert werden kann. Beendet werden kann das Framework an folgenden Stellen: von Extern (z.B. der

verwendenden Anwendung) und durch den RTCP-Thread (beim Empfangen eines `BYE`-Pakets). Daher muss dafür gesorgt werden, dass sowohl der aufrufende Code als auch der RTCP-Thread Zugriff auf die `stopAudioThreads()`-Methode des `OHMComm`-Objekts haben. Dafür wird dem RTCP-Thread ein Funktionsobjekt mitgegeben, das die `stopAudioThreads()`-Methode aufruft, die dafür sorgt, dass alle vom Framework erstellten Threads und Resource-Handler ordnungsgemäß beendet und geschlossen werden.
Somit kann die Kommunikation im OMHComm-Framework durch den Aufruf der `stopAudioThreads()`-Methode aus allen Stellen, die Zugriff darauf haben ordnungsgemäß und vollständig beendet werden. Jedoch wird die Kommunikation im Framework durch einen nicht blockierenden Methodenaufruf auf `startAudioThreads()` gestartet, damit der aufrufende Thread weitere Tätigkeiten ausführen kann. Da aber ein Programm automatisch beendet wird, wenn die `main()`-Funktion ihr Ende erreicht, muss für die prototypische Konsolenanwendung der Haupt-Thread nach dem Start der Kommunikation solange schlafen gelegt werden, bis die Kommunikation beendet wird. Dafür wird in der `main()`-Funktion blockierend auf eine Benutzereingabe gewartet und nach dieser Eingabe die Kommunikation und schließlich das Programm beendet. Wird jedoch die Kommunikation über einen anderen Weg (z.B. durch das Empfangen eines RTCP `BYE`-Pakets) beendet, wartet der Haupt-Thread weiterhin auf eine Benutzereingabe und somit läuft das Konsolenprogramm weiter. Deshalb muss derzeit die Konsolenanwendung immer mit einer Benutzereingabe (beliebige Zeichen, die mit einem „Enter“ abgesendet werden) beendet werden.

Kapitel 6

Schlussbemerkungen

6.1 Ausblick

6.1.1 Anwendungsmöglichkeiten

OHMComm kann als vollständiges und plattformunabhängiges Audiokommunikationsframework in anderen externen Softwarelösungen integriert werden. Als Schnittstelle nach außen bietet das Framework dafür ein `OHMComm`-Objekt an, welches erstellt, konfiguriert (bevorzugt über die `LibraryConfiguration`) und gestartet werden muss. Die gesamte Aktivität des Frameworks wird in separaten Threads (für Audioverarbeitung, Empfangen,, RTCP, ...) ausgelagert. So wird sichergestellt, dass die aufrufende Anwendung nicht blockiert wird. Durch vollständigen Funktionsumfang und wohldefinierten Schnittstellen lässt sich OHMComm einfach in andere Anwendungen integrieren. Kapitel 5 zeigt ein vollständiges Anwendungsbeispiel für die Audioübertragung zwischen zwei Geräten mit OHMComm. Am Beispiel lassen sich sehr einfach die verschiedenen Konfigurationsmodi und Einstellungsmöglichkeiten ausprobieren. Der Code dazu (bestehend aus der Datei `OHMCommStandalone.cpp`) zeigt ein kurzes und vollständiges Beispiel, wie das Framework richtig eingebunden werden kann.

6.1.2 Erweiterungsmöglichkeiten

Aufgrund des modularen Aufbaus des kompletten Frameworks, lässt es sich sehr einfach erweitern. Vorgesehene Erweiterungen für das Framework gliedern sich in die folgenden Kategorien:

Parameter: Alle existierenden Parameter können – soweit einmal registriert – von allen Audioprozessoren verwendet werden. Ebenso ist es sehr einfach, neue

Parameter zu registrieren, die bei der Konfiguration des Frameworks automatisch beachtet und mit Werten versehen werden.

Audioschnittstellen: Derzeit wird als einzige Schnittstelle zur Audiohardware die Bibliothek RTAudio verwendet. Jedoch kann die Audiobibliothek einfach ausgetauscht werden, indem eine neue Kindklasse von `AudioHandler` erstellt und zur `AudioHandlerFactory` hinzugefügt wird, die auf einer anderen Audiobibliothek aufbaut. So könnte z.B. auch PortAudio durch das Einfügen einer Wrapperklasse angebunden werden.

Audioprozessoren: In die Verarbeitungskette (siehe Abschnitt 3.3.3) können beliebige neue Prozessoren für Audiodaten hinzugefügt werden. Dafür muss nur die neu erstellte Kindklasse von `AudioProcessor` in `AudioProcessorFactory` registriert werden und beim Start der Anwendung ausgewählt werden. Ebenso können – wie bereits erwähnt – neue Parameter für die Konfiguration der neuen Audioprozessoren registriert werden.

Netzwerkschnittstellen: Auch neue Schnittstellen für Netzwerkprotokolle, wie TCP, lassen sich durch Erstellen neuer Kindklassen von `NetworkWrapper` und Ersetzen der Aufrufe von `UDPWrapper` hinzufügen.

Derzeit sind Erweiterungen zu dem bestehenden Framework in Planung oder bereits in Entwicklung, wie die Konfiguration über das Session Initiation Protocol (SIP), sowie den Audiocodecs G.711 A-law und μ-law, den Standardformaten der digitalen Telefonie.
Im Allgemeinen können in die bestehende Architektur eine Vielzahl an zusätzlichen Funktionalitäten fest oder auch optional hinzugefügt werden. So z.B. weitere Audiocodecs, Hochpass- oder Tiefpassfilter zum Herausfiltern von Rauschen oder Hintergrundgeräuschen, Verstärker für die Lautstärke des abgespielten Signals, Prozessoren, die die Audiokonversation mitschneiden und viele mehr.

6.2 Fazit

Das Hauptziel des Projektes war die Erstellung eines plattformunabhängigen Audiokommunikationsframework. Dieses Ziel wurde im Rahmen des IT-Projektes erreicht, da alle funktionalen und nicht-funktionalen Anforderungen umgesetzt worden sind. Es wurde eine abstrakte und austauschbare Schnittstelle zur Hardware geschaffen, welche ebenfalls die Architektur zur Audioverarbeitung beinhaltete (Verarbeitungskette). Dadurch konnten andere Klassen aktiv an der Audioverarbeitung teilnehmen. Die konkrete Implementierung dieser abstrakten Klasse basierte dabei auf RtAudio. Mit Opus wurde ein effizienter Audiocodec eingefügt, welcher die zuvor erstellte

Schnittstelle zur Verarbeitungskette nutzte. Das RTP-Protokoll wurde vollständig gemäß dem Standard implementiert und auf Basis des UDP-Protokolls versendet. Empfangene Pakete wurden im Jitter-Buffer sortiert eingefügt. Dieser puffert Pakete bis eine Mindestanzahl erreicht wurde, bevor diese zum Abspielen verfügbar waren. Alle erstellten Funktionalitäten wurden in einer Beispielanwendungen getestet. Die Statistikauswertung bestätigte die Effizienz des Frameworks sowie die Qualität des Opus-Codec. Obwohl das Projekt erfolgreich war gibt einige Kritikpunkte. Das Umsetzen der objektorientierten Prinzipien hätte konsequenter erfolgen können z.B. durch das Setzen der Konstruktoren auf private beim Factory-Method-Pattern. Zum Entwicklungszeitpunkt war dies jedoch nicht bekannt, da die entsprechende Vorlesung parallel dazu verlief. Ein weiterer Kritikpunkt war, dass der Code hätte sauberer sein können. Dies hätte durch das Festlegen von Code-Konventionen erfolgen können. Die Ursache hierfür war die mangelnde Praxiserfahrung des Teams. Trotz dieser Kritikpunkte bietet das Framework eine solide Grundlage für eine Weiterentwicklung, daher kann der Projektabschluss als Erfolg bewertet werden.

Literaturverzeichnis

[Goll and Dausmann, 2013] Goll, J. and Dausmann, M. (2013). *Architektur- und Entwurfsmuster der Softwaretechnik - Mit lauffähigen Beispielen in Java*. Springer.

[Lahres and Raýman, 2009] Lahres, B. and Raýman, G. (2009). *Objektorientierte Programmierung - Das umfassende Handbuch*. Galileo Computing, Bonn, 2. aufl. edition.

Online-Quellen

[Bencina, 2014] Bencina, R. (2014). Portaudio - an open-source cross-platform audio api. `http://www.portaudio.com/`. [letzter Zugriff: 9. Feb. 2016].

[CMake, 2016a] CMake (2016a). Cmake. `https://cmake.org/`. [letzter Zugriff: 9. Feb. 2016].

[CMake, 2016b] CMake (2016b). Cmake - license. `https://cmake.org/licensing/`. [letzter Zugriff: 9. Feb. 2016].

[Fraunhofer-IIS, 2016] Fraunhofer-IIS (2016). Aac-eld familie. `http://www.iis.fraunhofer.de/de/ff/amm/prod/kommunikation/komm/aaceld.html#tabpanel-3`. [letzter Zugriff: 9. Feb. 2016].

[GitHub, 2016] GitHub (2016). Github - where software is built. `https://github.com`. [letzter Zugriff: 9. Feb. 2016].

[Google, 2011] Google (2011). Issue 104241 - chromium - support opus in ogg files for the audio tag and audio object. `https://code.google.com/p/chromium/issues/detail?id=104241`. [letzter Zugriff: 9. Feb. 2011].

[IETF, 2003] IETF (2003). Rtp: A Transport Protocol for Real-Time Applications. `https://tools.ietf.org/html/rfc3550`. [letzter Zugriff: 24. Jan. 2016].

[IETF, 2015] IETF (2015). Webrtc audio codec and processing requirements. `https://tools.ietf.org/html/draft-ietf-rtcweb-audio-09#section-3`. [letzter Zugriff: 9. Feb. 2016].

[Lundell, 2013] Lundell, N. (2013). Cpptest - a c++ unit testing framework. `http://cpptest.sourceforge.net/`. [letzter Zugriff: 9. Feb. 2016].

[Microsoft, 2000] Microsoft (2000). Winsock reference. `https://msdn.microsoft.com/de-de/library/windows/desktop/ms740120%28v=vs.85%29.aspx`. [letzter Zugriff: 29. Jan. 2016].

[Microsoft, 2015a] Microsoft (2015a). Unterstützung für c++11/14/17-funktionen (modern c++). https://msdn.microsoft.com/de-de/library/hh567368.aspx. [letzter Zugriff: 9. Feb. 2016].

[Microsoft, 2015b] Microsoft (2015b). Visual studio - microsoft developer tools. https://www.visualstudio.com/. [letzter Zugriff: 9. Feb. 2016].

[Mozilla, 2016] Mozilla (2016). Media formats supported by the html audio and video elements - mdn. https://developer.mozilla.org/en-US/docs/Web/HTML/Supported_media_formats#Ogg_Opus. [letzter Zugriff: 9. Feb. 2016].

[OHMComm, 2016] OHMComm (2016). Github - ohmcomm license. https://github.com/doe300/OHMComm/blob/master/LICENSE. [letzter Zugriff: 9. Feb. 2016].

[Open-Handset-Alliance, 2014] Open-Handset-Alliance (2014). Android lollipop - android developers. http://developer.android.com/about/versions/lollipop.html. [letzter Zugriff: 9. Feb. 2016].

[Oracle, 2015] Oracle (2015). Netbeans. https://netbeans.org/. [letzter Zugriff: 9. Feb. 2016].

[Scavone, 2014a] Scavone, G. P. (2014a). The rtaudio home page. http://www.music.mcgill.ca/~gary/rtaudio/. [letzter Zugriff: 9. Feb. 2016].

[Scavone, 2014b] Scavone, G. P. (2014b). The RtAudio Home Page - API Notes. http://www.music.mcgill.ca/~gary/rtaudio/apinotes.html. [letzter Zugriff: 10. Jan. 2016].

[Scavone, 2014c] Scavone, G. P. (2014c). The rtaudio home page - license. http://www.music.mcgill.ca/~gary/rtaudio/license.html. [letzter Zugriff: 9. Feb. 2016].

[Skype, 2012] Skype (2012). Skype and a new audio codec - skype blogs. http://blogs.skype.com/2012/09/12/skype-and-a-new-audio-codec/. [letzter Zugriff: 9. Feb. 2016].

[TeamSpeak, 2013] TeamSpeak (2013). Teamspeak - teamspeak 3 client 3.0.10 released. http://forum.teamspeak.com/threads/84285-TeamSpeak-3-Client-3-0-10-released. [letzter Zugriff: 9. Feb. 2016].

[TravisCI, 2016] TravisCI (2016). Travis ci - test and deploy your code with confidence. https://travis-ci.org/. [letzter Zugriff: 9. Feb. 2016].

[Xiph.Org, 2011] Xiph.Org (2011). The celt ultra-low delay audio codec. http://celt-codec.org/. [letzter Zugriff: 9. Feb. 2016].

[Xiph.Org, 2015] Xiph.Org (2015). Opus codec 1.1 - api. https://www.opus-codec.org/docs/html_api-1.1.0/index.html. [letzter Zugriff: 9. Feb. 2016].

[Xiph.Org, 2016a] Xiph.Org (2016a). Opus codec. http://opus-codec.org/. [letzter Zugriff: 9. Feb. 2016].

[Xiph.Org, 2016b] Xiph.Org (2016b). Opus codec - license. http://opus-codec.org/license/. [letzter Zugriff: 9. Feb. 2016].