Hochschule für Technik und Wirtschaft Dresden
University of Applied Sciences

Fakultät Informatik/Mathematik

Bachelorarbeit

Thema:

Konzeption und prototypische Implementierung einer Erweiterung des NESSEE-Testsystems zur Unterstützung des Tests von Audiokomponenten

im Studiengang
Wirtschaftsinformatik

von

Thomas Meschke
Matrikelnummer 29543

betreut durch

Frau Prof. Dr. Anna Sabine Hauptmann
Herr Dipl. Inf. Robert Hess

bei der

Citrix Online Germany GmbH
Theaterstraße 6
01067 Dresden

eingereicht am
11.11.2013

Inhaltsverzeichnis

Abkürzungsverzeichnis

C	
COM	Component Object Model
H	
HTTP	Hypertext Transfer Protocol
I	
IPC	Inter-Process Communication
N	
NESSEE	Network Endpoint Server Scenario Emulation Environment
S	
SUT	Software Under Test
T	
TCP	Transmission Control Protocol
TDL	Test Case Description Language
TNM	TestNodeModule

Glossar

C

COM
Abkürzung für Component Object Model
Technik zur Erstellung von Programmiersprachen-unabhängigen Softwarekomponenten und zur Ermöglichung von → Inter-Process Communication. [Micro_3]

D

Deserialisierung
Umkehroperation der → Serialisierung

F

Framework
deutsch: Rahmenwerk
Programmgerüst, welches selbst noch kein fertiges Programm darstellt, jedoch die grundlegende Architektur und den Kontrollfluss vorgibt. Kann durch konkrete Implementierungen zu einem Programm vervollständigt werden. Ziel ist die Erreichung eines hohen Grades der Wiederverwendbarkeit von häufig genutzten Mustern. [ItWissen_1]

G

Google Protocol Buffers
Sprachunabhängige, plattformübergreifende Technik zur → Serialisierung strukturierter Daten. Google selbst beschreibt die Technologie derart: „Denken Sie an XML, nur kleiner, schneller und einfacher“ ([Google_1]).

I

Inter-Process Communication
deutsch: Interprozesskommunikation
Kommunikation, also Informations- und Datenaustausch, zwischen nebenläufigen Prozessen auf dem selben Rechner, deren Speicherbereiche voneinander getrennt sind. [ItWissen_2]

L

Log
auch:Logdatei, Log file, Protokolldatei
Automatisch geführte Protokolldatei aller oder bestimmter Aktionen von Prozessen auf einem Computersystem. [ItWissen_3]

M

Meta-Informationen
auch: Meta-Daten
Informationen über Nutzdaten, jedoch nicht diese Daten selbst. Dienen zur Verwaltung, Archivierung oder das Management von Daten und enthalten Angaben wie die Größe, Struktur oder das Format dieser Daten. [ItWissen_4]

N

NESSEE
Akronym, steht für „Network Endpoint Server Scenario Emulation Environment“, eine Softwareumgebung zur Durchführung automatisierter Softwaretests.

R

Router
Netzwerkgerät, welches Netzwerkpakete zwischen verschiedenen Rechnernetzen weiterleiten kann und somit bspw. eine Kopplung mehrerer Standorte ermöglicht. [ItWissen_5]

S

Subnetz
Bezeichnung für einen Teil eines größeren Computer-Netzwerkes auf Basis des Internetprotokolls. Ermöglicht z.B. die Zusammenfassung mehrerer PCs nach ihrem Standort. [ItWissen_6]

Script-Engine
Allgemeine Bezeichnung für Programme und Bibliotheken, welche die Steuerung einer Anwendung oder ihres Verhaltens mithilfe sogenannter Scripte ermöglichen.

Serialisierung
Umwandlung von Anwendungsdaten in ein sequenzielles Datenstromformat zur Speicherung oder verlustfreien Übertragung.

S (Forts.)

SUT
Abkürzung für Software Under Test
Eigenständige Software-Anwendung, welche die für Tests benötigten Kernfunktionalitäten und die zugehörige Geschäftslogik des späteren Produktes enthält.

W

Wrapper
Bezeichnet in der Softwaretechnik ein Stück Software, welches in der Funktion eines Adapters ein anderes Stück Software umgibt, beispielsweise aus Gründen der Kompatibilität, Sicherheit oder des Zugriffsschutzes. [ItWissen_7]

X

XML
Abkürzung für Extensible Markup Language
Metasprache, welche der hierarchischen Darstellung strukturierter Daten in textueller Form dient.

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Grundlagen

Citrix Online, die Online Services Division (OSD) der Citrix Systems Inc. verfügt über einen stetigen Bedarf an Tests für Client- und Server-Software. Bei diesen Tests werden die Produkte sowohl auf eine möglichst geringe Fehlerquote im Hinblick auf ihren funktionalen Umfang, als auch auf ihr zuverlässiges Verhalten unter hohen Anforderungen, wie beispielsweise großer Serverlast, getestet.

Um den Workflow der Produktentwicklung und -erstellung nicht unnötig aufzuhalten, kommt der Automatisierung dieser Tests direkt nach einer Änderung der Software eine herausragende technische, als auch wirtschaftliche Bedeutung zu. Der anfänglich vergleichsweise höhere Aufwand zur Entwicklung solcher automatisierter Tests sichert die Erreichung gleichbleibender Qualität und minimiert in der Folge die Kosten für jeden weiteren Testlauf. Zu diesem Zweck entstanden im Zeitraum der letzten Jahre zahlreiche kleine und große Frameworks, Suiten und andere Softwareprodukte, welche das Aufgabenfeld der Testautomatisierung in unterschiedlich großem Rahmen abdecken. Nach der Betrachtung zahlreicher dieser Testumgebungen wurde deutlich, dass alle von ihnen gewisse Einschränkungen aufweisen. Beispielsweise wenn es darum geht, Szenarien mit mehr als 1 000 teilnehmenden Endpunkten abzubilden, oder den Einfluss eines vielseitigen, komplexen Netzwerkes in den Test einfließen zu lassen.

Aus diesem Grund entwickelte die Citrix Online Video-Gruppe in Dresden in Kooperation mit der Technischen Universität Dresden ein Projekt mit dem Namen „NESSEE – Network Endpoint Server Scenario Emulation Environment“. Das Ziel dieses Projektes ist es, eine Umgebung zu schaffen, in welcher das Verhalten von komplexen Netzwerken emuliert werden kann. Mithilfe dieser Umgebung können reproduzierbare Tests von verteilten Anwendungen bezüglich ihrer Funktionalität und ihrer Skalierbarkeit durchgeführt werden.

In einem ersten Schritt wurde bisher ein Framework geschaffen, welches das automatisierte Testen der kompletten Video-Infrastruktur von einem Endpunkt zu einem anderen ermöglicht. Die Umsetzung als ein verteiltes System ermöglicht hierbei die Skalierung des Testsystems in Abhängigkeit vom aktuellen Bedarf an Ressourcen aus den Bereichen Prozessorleistung, Arbeitsspeicher- oder Festplattenplatz und Netzwerkleistung.

Mit dem Einsatz einer Netzwerk-Emulation können dabei Aspekte wie beispielsweise unterschiedliche Bandbreiten, Übertragungsverzögerungen oder Paketverlust, sowie deren Einfluss auf die Funktion, das Verhalten und die Zuverlässigkeit der zu testenden Software analysiert werden. Die grundlegende Architektur des Systems in seinem aktuellen Entwicklungsstand ist der **Abbildung 1** zu entnehmen.

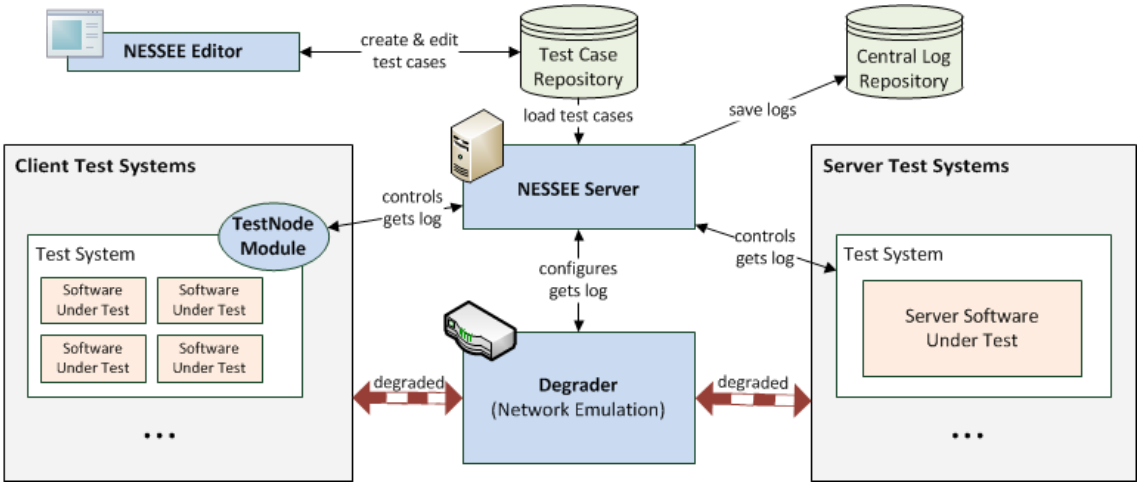


Abbildung 1: Derzeitige Architektur des NESSEE-Systems (vgl. [LÜBKE u.a. 2012])

Die zentrale Komponente bildet hierbei der *NESSEE Server*, dessen Zuständigkeit in der Koordination der beteiligten Entitäten liegt. Die Testsysteme sind überwiegend, jedoch nicht zwingend, virtuelle Maschinen (VM), welche auf einer Vielzahl von Servern ausgeführt werden. Dabei werden im Wesentlichen zwei Arten von Testsystemen unterschieden:

- *Client-Testsysteme* – Derzeit VMs auf Basis von Microsoft Windows, auf welchen mehrere Instanzen der zu testenden Software parallel ausgeführt werden. Diese stellen die Benutzer-Endpunkte der verteilten Anwendung dar.

- *Server-Testsysteme* – Derzeit VMs auf Basis von Red Hat Enterprise Linux, auf welchen die Serverkomponenten des Video-Konferenzsystems ausgeführt werden, die sogenannten Videocluster. Diese übernehmen die Signal- und Datenstromverarbeitung zwischen den Teilnehmern einer Konferenz.

Die zur Verfügung stehenden Testfälle und deren Konfiguration erhält der NESSEE Server dabei aus dem *Test Case Repository,* einem zentralen Speicher für Testfallbeschreibungen. Diese Beschreibungen setzen sich aus mehreren Dateien in XML sowie JavaScript-Dateien zusammen. Sie dienen zum Steuern von Abläufen auf Client- und Serverseite. Um ein hohes Maß an Flexibilität bieten zu können, sind die Testfallbeschreibungen sehr umfangreich. Ein eigens dafür entwickelter Editor hilft dabei, diese zu erstellen.

Das manuelle Verfassen mithilfe eines Texteditors würde wesentlich längere Zeit in Anspruch nehmen. Außerdem birgt es eine große Fehlerquelle, welche auf diese Art neutralisiert wird. Darüber hinaus wäre in jedem Fall ein hohes Maß an Wissen über Syntax und Semantik der Test Case Description Language (TDL) nötig.

Eine weitere wichtige und zentrale Komponente ist der *Degrader*, dessen Aufgabe die Emulation des Netzwerkes ist. Die Testsysteme für Server und Clients sind in unterschiedlichen sogenannten Subnetzen angeordnet. Der Degrader fungiert dabei als Router für alle involvierten Testsysteme und wird vom NESSEE Server auf Basis der Testfallbeschreibung konfiguriert. Auf dieser Grundlage werden einzelne Verbindungen künstlich manipuliert, um Paketverlust, schwankende Bandbreiten, Verzögerungen in der Übertragung, Duplizierung von Paketen oder die Änderung ihrer Reihenfolge sowie ihre Beschädigung deterministisch und wiederholbar abbilden zu können. Des Weiteren können mithilfe des Degraders verschiedene Einflüsse, beispielsweise Hintergrund-Traffic, sich während der Übertragung ändernde Routen oder verschiedene Netzwerktopologien, emuliert werden.

Das *TestNodeModule (TNM)* ist ein Systemdienst, welcher auf jedem Client-Testsystem genau einmal läuft. Aufgabe des TNM ist es, einen clientseitigen Einstiegspunkt für die Kommunikation mit dem NESSEE Server bereit zu stellen. Über diesen werden einerseits Befehle vom Server zu den Testsystemen übertragen, um Handlungsanweisungen an die Software Under Test (SUT) zu geben. Andererseits werden hierüber Protokolldateien (Logs) und statistische Daten zum Server übertragen. Hierbei handelt es sich um Daten zum Verhalten der Anwendungen und des Testsystems, wie beispielsweise die aktuelle CPU-Auslastung. Auf jedem der Testsysteme können darüber hinaus mehrere Instanzen der SUT durch das TNM gestartet, verwaltet und gesteuert werden. Auf diese Weise werden Tests stets auf so wenigen Maschinen wie möglich ausgeführt, sodass die zur Verfügung stehenden Ressourcen optimal genutzt werden können.

1.2 Zielstellung

Im Rahmen dieser Arbeit soll der funktionale Umfang des NESSEE-Systems so erweitert werden, dass auch Audiokomponenten auf dieselbe Art und mit einer ebenso großen Flexibilität getestet werden können, wie bisher die Videokomponenten. Wichtige Fragen sind hierbei, welche Anpassungen des bisherigen Systems nötig sind, um die Anforderungen umsetzen zu können. Auch die Wiederverwendbarkeit von Teilen des bisherigen Entwicklungsstandes ist zu prüfen. Die durch die Testsoftware zur Verfügung gestellten Funktionen werden dabei durch eine Schnittstelle beschrieben, welche von einem anderen Entwicklungsteam bereitgestellt wird.

Im Fokus der zu konzipierenden Lösung liegen dabei die folgenden Teilaspekte:

1. ***Entwicklung einer Kommunikationstechnik auf Basis der zur Verfügung gestellten Schnittstellenbeschreibung zur Ermöglichung der Kommunikation des TestNodeModule mit der Software Under Test***

Die Herausforderung liegt hierbei im vorliegenden technologischen Bruch. Die Schnittstellenbeschreibung und deren Implementierung liegen in C++ vor, das TNM ist auf Basis des .Net-Frameworks in C# realisiert. Es besteht die Möglichkeit, unter C++ erstellte Klassenbibliotheken in einem C#-Programm zu referenzieren und zu verwenden. Da eine direkte Kommunikation der beiden Komponenten jedoch eine zu enge Bindung und Abhängigkeit schaffen würde, muss eine andere Möglichkeit der Interprozesskommunikation (IPC) gefunden werden.

2. ***Anpassung und Erweiterung der verwendeten JavaScript-Bibliotheken, um die neuen Funktionen im Rahmen der Testausführung nutzbar zu machen***

Das Verhalten der Testanwendung während der Testausführung wird mithilfe von JavaScript-Dateien gesteuert. Viele der derzeit verwendeten videospezifischen Funktionen werden für Audiotests in gleicher oder ähnlicher Weise benötigt. Um die Dopplung von Quelltext zu vermeiden ist zu überprüfen, inwieweit die vorhandenen Funktionen abstrahiert werden können. So werden diese für beide Testarten nutzbar gemacht. Darüber hinaus werden auf diese Weise die Wartbarkeit und Wiederverwendbarkeit erleichtert.

3. ***Anpassung der Spezifikation der Testfallbeschreibungen zur Unterscheidung zwischen einem Video- und einem Audiotest***

Die an einem Videotest beteiligten Komponenten (Endpunkte, Server, Degrader, …) werden zuvor in der TDL anhand definierter Parameter konfiguriert. Hier ist zu überprüfen, welche dieser Parameter geändert und ob neue hinzugefügt werden müssen, um Audiotests durchführen zu können.

4. ***Eventuelle Erweiterung der Kommunikationsmöglichkeiten des Servers mit dem TNM, um die neuen Funktionen für die Verwendung in Testfallbeschreibungen verfügbar zu machen***

Die Übertragung sowohl der Konfiguration als auch der Testanweisungen vom Server zu den teilnehmenden Testsystemen erfordert ein Protokoll, welches beide Seiten gleichermaßen beherrschen. Ein solches Protokoll wurde bereits für die Ausführung von Videotests definiert. Zur Durchführung von Audiotests ist es möglicherweise nötig, dieses Protokoll um neu hinzukommende Funktionen und Anweisungen zu erweitern.

5. ***Erweiterung der bisherigen Kommunikationsschnittstelle des TNM, um zusätzlich benötigte Funktionalitäten bereitstellen zu können***

Um den Funktionsumfang der Schnittstellenbeschreibung nutzen zu können, ist es nötig, auch zwischen TNM und SUT ein Protokoll zu definieren. Des Weiteren muss der durch die Schnittstelle beschriebene Einstiegspunkt der SUT um einen Kommunikationsrahmen erweitert werden, welcher die Kontaktaufnahme durch das TNM ermöglicht. Die Video-Testanwendung stellt zu diesem Zweck einen COM-Wrapper bereit. Auf einen solchen soll jedoch bei dieser Erweiterung verzichtet werden, da die Verwendung von COM nicht robust genug ist und nicht hinreichend skaliert.

6. ***Erweiterung des Software-Verteilungsprozesses um die Übertragung der Audio-Testsoftware vom Server zu ausgewählten Endpunkten***

Da die Testanwendung, welche für Audiotests zum Einsatz kommen soll, nicht automatisch auf den verwendeten Endpunkten vorhanden ist, muss es eine Möglichkeit geben, diese ausgehend vom Server zu verteilen. Für die vorhandene Video-Testanwendung ist ein solchen Verfahren bereits realisiert, sodass lediglich eine Erweiterung vorzunehmen ist.

2 Analyse des aktuellen Systems

Um eine Grundlage zu schaffen, auf welcher das Konzept der Erweiterung erstellt werden kann, wird an dieser Stelle zunächst skizziert, wie das derzeitige System aufgebaut ist. Dazu wird im Folgenden der Ablauf der Abarbeitung einer Testanweisung in ihren einzelnen Etappen erläutert sowie alle involvierten Komponenten erklärt.

Das System zum Testen der Videokomponenten bedient sich eines Modells auf Basis von Konferenzen und Nutzern. Ein Nutzer ist hierbei immer ein Teilnehmer einer Konferenz, und eine Konferenz kann prinzipiell beliebig viele Teilnehmer, also Nutzer, enthalten. Insbesondere kann eine solche Konferenz auch ohne Nutzer existieren. Um einen Test durchführen zu können, muss daher zuerst eine Konferenz erstellt werden. Dies geschieht durch eine Anfrage an einen Server, welcher Teil der Videocluster-Architektur ist. Auf die konkrete Architektur des Videokonferenzsystems soll an dieser Stelle nicht weiter eingegangen werden, da dies nicht Gegenstand der vorliegenden Arbeit ist. Nach der erfolgreichen Erstellung der Konferenz können sich die Nutzer in diese einloggen, um ihr beizutreten. An dieser Stelle werde ich die Erläuterung des Testablaufes beginnen. Dazu sei zunächst auf **Abbildung 2** verwiesen.

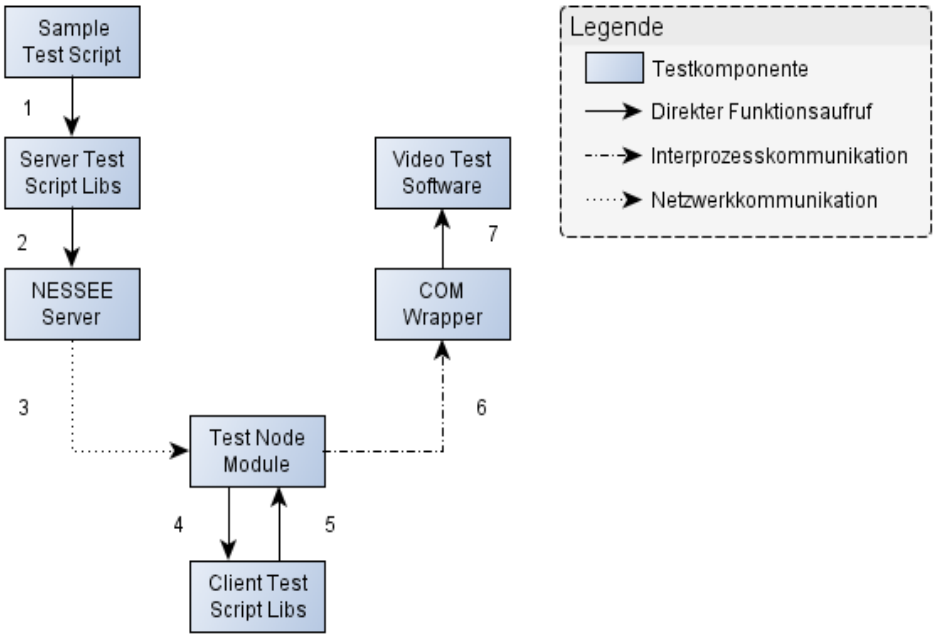


Abbildung 2: Etappen und Komponenten der Testausführung

Die Beschreibung des Tests erfolgt auf Basis von JavaScript. Dieses Testscript enthält die auszuführenden Anweisungen. In Abbildung 2 ist dies durch den Kasten „Sample Test Script“ verdeutlicht. Häufig wiederkehrenden Funktionen, welche in diesem Script verwendet werden können, sind in den sogenannten „Test Script Libs“ definiert. Dabei handelt es sich um eine Sammlung vorbereiteter JavaScript-Dateien, welche in die eigentlichen Testscripte eingebunden werden. Unterteilt wird dabei in Funktionen, die serverseitig zum Einsatz kommen, und solchen, die clientseitig benötigt werden. Um diese Einteilung zu verdeutlichen ist in Abbildung 2 die Rede von „Server Test Script Libs“ und „Client Test Script Libs“, auch wenn es sich hierbei um ein und dieselbe Sammlung von Funktionen handelt.

Die Server-Scriptbibliothek enthält eine Funktion mit Namen „user_login“, welche das Anmelden eines Benutzers an einer bestehenden Konferenz ermöglicht. An dieser Funktion wird der Ablauf beispielhaft beschrieben. Die Funktion wird aus dem Testscript aufgerufen (Abb. 2, Nr. 1) und übernimmt beispielsweise die Überprüfung, ob der ausgewählte Benutzer bereits in der Konferenz angemeldet ist. So kann bereits an dieser Stelle die doppelte Ausführung des Anmeldevorgangs vermieden werden. Auf diese Weise lassen sich Netzwerk- und Prozessorressourcen sparsam einsetzen.

Mithilfe des Microsoft Script Controls [vgl.(Micro_1)], einer Script-Engine, die zur Ausführung der Scripte auf Basis von JavaScript benötigt wird, können aus der Scriptbibliothek Aufrufe von Funktionen der NESSEE Server Komponente durchgeführt werden (Abb. 2, Nr. 2). In diesem Fall gibt es jedoch seitens des NESSEE Servers nicht für jede der Scriptfunktionen eine entsprechende Zielfunktion. Vielmehr gibt es nur einige wenige Funktionen, welche die clientseitige Ausführung einer durch ihren Namen identifizierten Funktion ermöglicht. Hierbei kann die Ausführung für alle angemeldeten Benutzer, oder für einen bestimmten Benutzer erfolgen. Im aktuellen Beispiel erfolgt ein Aufruf der Serverfunktion „ExecuteForOneUser“. Dieser wird der betreffende Nutzer sowie namentlich die auf Clientseite auszuführende Funktion – in diesem Fall „client_login“ – sowie deren Parameter übergeben.

Auf Basis dieser Informationen sendet der NESSEE Server über eine bestehende Netzwerkverbindung eine Nachricht an ein verbundenes TestNodeModule (Abb. 2, Nr. 3). Die Kommunikation basiert auf Google's „Protocol Buffers“, einer sprach- und plattformunabhängigen Möglichkeit zum Austausch von zuvor definierten Nachrichten (vgl. [Google_1]).

Nach dem Empfang dieses Kommandos, dessen Inhalt der betreffende Nutzer sowie die aufzurufende Funktion samt ihrer Parameter sind, führt das TNM seinerseits einen Aufruf der angegebenen Funktion durch (Abb. 2, Nr. 4). Diese Funktion, hier „client_login“, ist selbst in den bereits angesprochenen „Client Test Script Libs“ definiert.

Durch das Microsoft Script Control wird an dieser Stelle wiederum ein Aufruf von Funktionen des TNMs aus den clientseitigen Testscripten ermöglicht (Abb. 2, Nr. 5). Zu diesem Zweck definiert das TestNodeModule eine Reihe von Funktionen, welche den Funktionsumfang der eigentlichen Videotestsoftware widerspiegeln. Seitens der Testanwendung werden diese Funktionen durch weiter oben angesprochenen COM-Wrapper gekapselt und bereitgestellt.

Die Testanwendung ist in C++ implementiert, das TNM basiert auf C#, einer Sprache aus dem Microsoft .Net Framework. Da der direkte Aufruf von C++-Funktionen aus einem C#-Quelltext nur unter größerem Aufwand zu erreichen ist und zudem eine sehr enge Bindung zwischen den betreffenden Komponenten erzeugt (vgl. [Micro_2]), wurde an dieser Stelle auf die Kommunikation mittels der COM-Technologie gesetzt (vgl. [Micro_3]).

Mithilfe des COM-Wrappers werden die Funktionen der Testanwendung gekapselt und nach außen verfügbar gemacht. So können, über einen Aufruf einer Funktion dieses COM-Wrappers (Abb. 2, Nr. 6), die eigentlichen Funktionen der Testanwendung ausgeführt werden (Abb. 2, Nr. 7). Diese enthält schließlich die eigentliche Funktionalität, im aktuellen Beispiel das Anmelden eines Nutzers an einer Konferenz.

An dieser Stelle stellt sich möglicherweise die Frage, warum auf Clientseite der scheinbare Umweg über die Scriptbibliothek gegangen wird, anstatt die entsprechende Funktion direkt im TNM bereit zu stellen und auszuführen. Diese Aufteilung bietet den Vorteil, eine gewisse Granularität und Modularität der Funktionen zu schaffen. So ist es mithilfe der Client-Testscripte beispielsweise möglich, einzelne Kernfunktionen zu bündeln, wenn diese häufig zusammen ausgeführt werden. Andererseits können durch diese Aufteilung einzelne Bestandteile geändert oder ausgetauscht werden, ohne andere Teile zu beeinflussen. Diese feine Unterteilung ist ein wesentlicher Grund für die hohe Flexibilität des Systems.

3 Anforderungen

Um eine vollständige und überschneidungsfreie Zuordnung vornehmen zu können, werden die Anforderungen in die folgenden drei Arten unterteilt:

- Funktionale Anforderungen
- Qualitätsanforderungen und
- Rahmenbedingungen

(vgl. [POHL 2008])

Dabei ist jedoch zu beachten, dass es sowohl Anforderungen gibt, welche speziell für die im Rahmen dieser Arbeit zu konzipierende Erweiterung des NESSEE-Systems gelten, als auch solche, die für das NESSEE-System insgesamt gelten. Es kann davon ausgegangen werden, dass letztere im aktuellen Entwicklungsstand bereits umgesetzt sind. Hierbei ist dafür Sorge zu tragen, dass diese Anforderungen auch nach der Implementierung der Erweiterung auf Grundlage der Konzeption als erfüllt betrachtet werden können.

Alle Anforderungen, welche speziell für die Integration der Erweiterung gelten, dürfen somit nicht im Konflikt zu den bereits bestehenden Anforderungen stehen, sondern sind diesen unterzuordnen. Auf dieser Grundlage lassen sich die in **Tabelle 1** gezeigten funktionalen Anforderungen festlegen. Die **Abbildungen 3-6** zeigen die zugrunde liegenden Anwendungsfälle. **Tabelle 2** listet die Qualitätsanforderungen auf. In **Tabelle 3** sind die Rahmenbedingungen zu finden.

Funktionale Anforderungen			
Nr.	**Beschreibung**		**Geltungsbereich**
FA 1	Testfallbeschreibungen können verwaltet werden.		NESSEE
	FA 1.1	Testfallbeschreibungen können erstellt werden.	
	FA 1.2	Vorhandene Testfallbeschreibungen können bearbeitet werden.	
	FA 1.3	Vorhandene Testfallbeschreibungen können gelöscht werden.	
	FA 1.4	Vorhandene Testfallbeschreibungen können in Testsequenzen integriert werden.	
FA 2	Testläufe können durchgeführt werden.		NESSEE
	FA 2.1	Testläufe können auf Basis einer zugeordneten Testfallbeschreibung gestartet werden.	
	FA 2.2	Mit gestarteten Testläufen kann interagiert werden.	
	FA 2.3	Gestartete Testläufe können unterbrochen werden.	
	FA 2.4	Testergebnisse können ausgewertet werden, nachdem ein Testlauf beendet wurde.	
FA 3	Testsequenzen können verwaltet werden.		NESSEE
	FA 3.1	Testsequenzen können erstellt werden.	
	FA 3.2	Testsequenzen können bearbeitet werden.	
	FA 3.3	Testsequenzen können gelöscht werden.	
FA 4	Alle durch die SUT bereitgestellten Funktionen werden für die Verwendung in Testscripten verfügbar gemacht.		Audio-Integration

Tabelle 1: Funktionale Anforderungen an das zu erstellende System

Qualitätsanforderungen			
Nr.	**Beschreibung**		**Geltungsbereich**
QA 1	Die Verarbeitung von Benutzereingaben/-aktionen soll auch im Mehrbenutzerbetrieb keine Einschränkungen der Benutzbarkeit verursachen.		NESSEE
QA 2	Die Performanz des Systems ist hinsichtlich dem Datendurchsatz zu optimieren, um durch möglichst viele gleichzeitige Anweisungen ein Maximum an Last erzeugen zu können.		NESSEE
QA 3	Das System darf nicht aufgrund von Falscheingaben abstürzen.		NESSEE
QA 4	Tests müssen reproduzierbar sein.		NESSEE
QA 5	Fehlertoleranzmaßnahmen sind zu ergreifen.		NESSEE
	FA 4.1	Auf auftretende Fehler ist durch Meldungen aufmerksam zu machen, ein Testlauf ist zu beenden.	
	FA 4.2	Aufgetretene Fehler sind in einer Protokolldatei (Log) zu speichern.	

Tabelle 2: Qualitätsanforderungen an das zu erstellende System

Rahmenbedingungen		
Nr.	**Beschreibung**	**Geltungsbereich**
RB 1	Ausgaben, Benutzerführung und Oberfläche werden in englischer Sprache umgesetzt.	NESSEE
RB 2	Dokumentation(en) und Quelltextkommentare sind in englischer Sprache zu verfassen.	NESSEE
RB 3	Die Lösung ist in das bestehende System zu integrieren und muss daher mit dem Microsoft .Net Framework Version 3.5 oder höher lauffähig sein.	Audio-Integration
RB 4	Bereits integrierte Lösungen zur Protokollierung, zur Kommunikation und zur Fehlerbehandlung sind möglichst weiter zu verwenden.	Audio-Integration
RB 5	Die Kommunikation zwischen den einzelnen Komponenten erfolgt asynchron.	Audio-Integration

Tabelle 3: Rahmenbedingungen, denen das zu erstellende System unterliegt

Der Zusammenhang zwischen den funktionalen Anforderungen soll im Folgenden durch Anwendungsfall-Diagramme verdeutlicht werden.

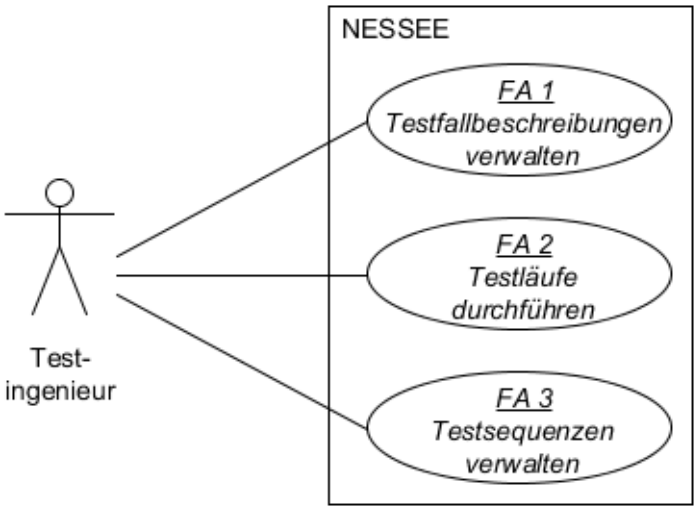


Abbildung 3: Abstrakte Anwendungsfälle, vgl. FA 1 - FA 3

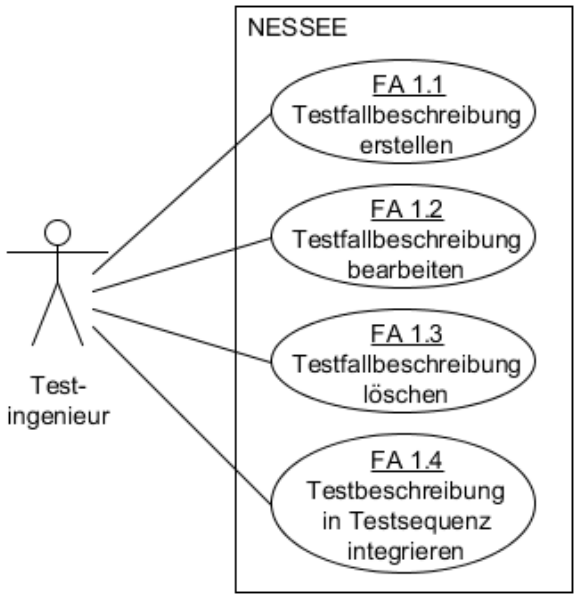


Abbildung 4: Konkrete Anwendungsfälle aus FA 1

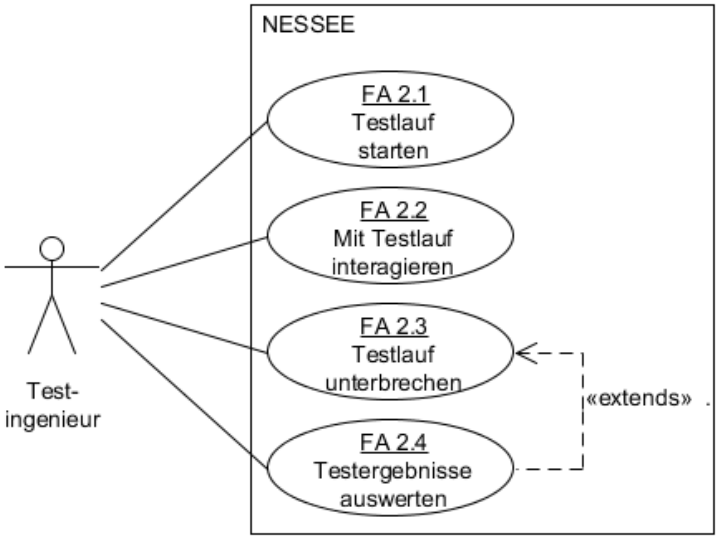


Abbildung 5: Konkrete Anwendungsfälle aus FA 2

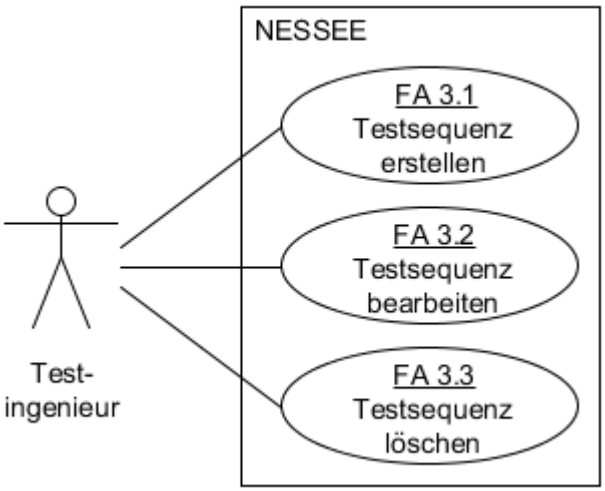


Abbildung 6: Konkrete Anwendungsfälle aus FA 3

Testsequenzen stellen in diesem Zusammenhang statische Strukturen dar. Sollen mehrere Testläufe aufeinander folgend ausgeführt werden, können die zugehörigen Testfallbeschreibungen in der entsprechenden Reihenfolge zu einer Testsequenz zusammengefasst werden. Auf diese Weise ist das automatische Ausführen mehrerer Testläufe möglich. So können beispielsweise größere Szenarien abgebildet werden, ohne dabei auf die Granularität einzelner Testfälle verzichten zu müssen.

Da es sich nur um Strukturen zur hierarchischen Ordnung von vorhandenen Testfallbeschreibungen handelt, verfügen die Testsequenzen über keine eigene Dynamik. Das Ausführen einer solchen Testsequenz zerfällt folglich in das aufeinander folgende Ausführen aller ihr zugeordneten Testfallbeschreibungen als separaten Testlauf.

4 Entwurf des Soll-Systems

4.1 Vorüberlegungen

Nachdem die Elemente des Systems näher beleuchtet wurden, kann auf dieser Grundlage das Konzept für die Erweiterung geschaffen werden.

Im Grunde genommen verläuft ein Audiotest nicht anders als ein Videotest. Das Modell von Konferenzen und Nutzern findet nahezu unverändert Anwendung, jedoch besteht die Cluster-Achitektur in diesem Fall aus Servern, welche auf die Verarbeitung von Audiosignalen spezialisiert sind.

Eine wesentliche Änderung ergibt sich einzig in der Art der Interprozess-kommunikation auf der Seite der Testsoftware. Die Verwendung von COM, wie bisher bei der Durchführung von Videotests, war zu vermeiden. Die Nachteile dieser Technik offenbaren sich erst bei genauerer Untersuchung und Verwendung. Das Component Object Model wurde als Black-Box-Lösung konzipiert. Das heißt, dass in vielen Fällen nicht ersichtlich ist, wie die Verarbeitung von Funktionen intern abläuft. Im Falle eines Fehlers macht dieses Verhalten die Fehlersuche ausgesprochen schwer. Darüber hinaus wurde festgestellt, dass es Probleme mit dem Lebenszyklus von COM-Objekten bei deren Verwendung durch .Net gibt. So werden beispielsweise Referenzen auf nicht mehr benötigte Objekte gar nicht oder erst sehr spät aus dem Speicher entfernt. Dieses Verhalten beeinflusst nicht nur die Reproduzierbarkeit der Tests, es eröffnet auch Probleme mit den begrenzten Ressourcen. Auf diese Weise entstehende Speicherlecks gefährden die Skalierbarkeit der Tests und somit den gesamten Workflow.

Aufgrund der guten Erfahrungen, welche mit Google's Protocol Buffers gesammelt wurden, wurde entschieden, auch die Kommunikation zwischen TNM und der Testsoftware auf dieser Technik aufzubauen. Der Vorteil ist die deutliche Plattformunabhängigkeit eines auf offenen Standards und Open Source Software basierenden Kommunikations-Protokolls. Diese Änderung findet sich auch in **Abbildung 7** wieder.

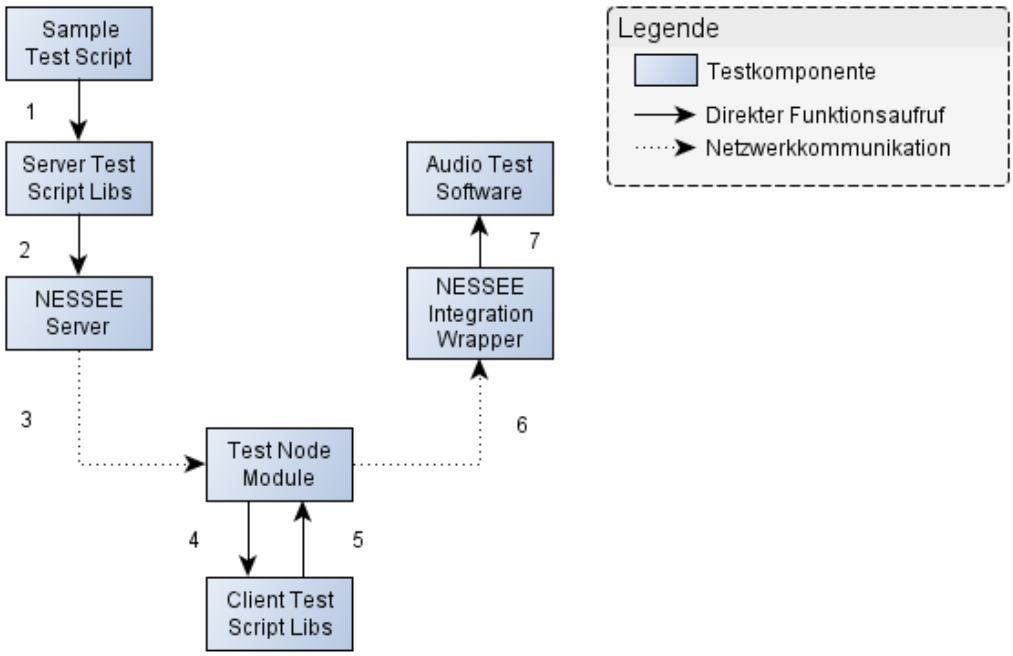


Abbildung 7: Etappen und angepasste Komponenten der Audio-Integration

Erkennbar ist, dass die vormalige Interprozesskommunikation durch eine Netzwerkkommunikation abgelöst wird. Dies bedeutet jedoch keinesfalls, dass sich das TNM und die Testsoftware nun auf entfernten Systemen befinden. Es bildet lediglich die zuvor erläuterte Realisierung der Interprozesskommunikation durch den Einsatz der Protocol Buffers als Netzwerkprotokoll ab. Vereinfacht ausgedrückt stellt das Testsystem an dieser Stelle sowohl einen Server als auch einen Client dar. Hierbei bildet die Testsoftware entweder den Client-Prozess, welcher sich zum TNM in der Funktion des Server-Prozesses verbindet, oder umgekehrt. Die Flexibilität wird durch diese Umsetzung weiter erhöht, da eine Plattformunabhängigkeit der Kommunikation erreicht wird. Selbst die physische Trennung des TNM und der Software Under Test wird prinzipiell ermöglicht, auch wenn letzteres in absehbarer Zeit keine Anwendung finden wird.

Die umzusetzenden Anpassungen und Entwicklungen erschließen sich nun, wenn man die oben aufgezeigten Etappen in umgekehrter Reihenfolge betrachtet. Die Testsoftware stellt Funktionen bereit, welche dem TNM zur Verwendung zugänglich gemacht werden müssen. Diese Aufgabe wird dem NESSEE Integration Wrapper zuteil, welcher die bereitgestellten Funktionen kapselt und eine Kommunikations-Schnittstelle für das TNM auf Basis der Protocol Buffers bereitstellt.

Die so zur Verfügung gestellten Funktionen müssen durch die Auswahl an interpretierbaren Befehle durch das TNM unterstützt werden. Somit ist eine Erweiterung des Umfangs der bekannten Befehle notwendig. Das Absetzen dieser neuen Kommandos kann lediglich durch den Aufruf aus den „Client Test Script Libs“ erfolgen, sodass diese ebenso um entsprechende Funktionen erweitert werden müssen.

Die Kommunikation zwischen dem NESSEE Server und dem TNM ist bereits sehr generisch gehalten, da lediglich der Name der auszuführenden Funktion und deren Parameter übertragen werden. Somit sollten keine oder nur geringe Anpassungen nötig sein, um die clientseitige Ausführung der hinzukommenden Funktionen zu ermöglichen.

Die „Server Test Script Libs“ sind in ihrem Umfang zu erweitern, um die neuen Funktionen für die Verwendung in den eigentlichen Testscripten verfügbar zu machen.

Anhand eines einfachen Beispiel-Testscriptes ist abschließend sicher zu stellen, dass die Abarbeitung in allen Komponenten korrekt erfolgt und sich die Testsoftware gemäß der im Testscript festgelegten Abläufe verhält.

4.2 Konzeption der Umsetzung

Zunächst wird an dieser Stelle etwas genauer auf den Aufbau der Audio-Testsoftware eingegangen. Wie in **Abbildung 8** zu sehen, besteht diese aus mehreren einzelnen Komponenten, von denen im wesentlichen zwei von besonderer Bedeutung sind:

- Die Audio Endpoint Engine und
- Die TestLib mit dem HTTP-Connector

Die Audio Endpoint Engine enthält dabei alle Funktionen, welche auch im späteren Produktivsystem Anwendung finden und getestet werden sollen, und bildet damit das funktionale Fundament der gesamten Anwendung.

Auf derselben Ebene findet sich die sogenannte TestLib, eine Klassenbibliothek, welche alle Funktionen bereitstellt, die für das Durchführen von Tests nötig sind. Hier wird beispielsweise alle Anwendungslogik abgebildet, welche im Produktivsystem sonst von anderen Komponenten gehandhabt wird.

Dabei wird auf strenge Trennung der Verantwortlichkeiten geachtet, sodass sich in der TestLib nur Funktionalitäten befinden, welche tatsächlich im Rahmen der Testausführung von Bedeutung sind.

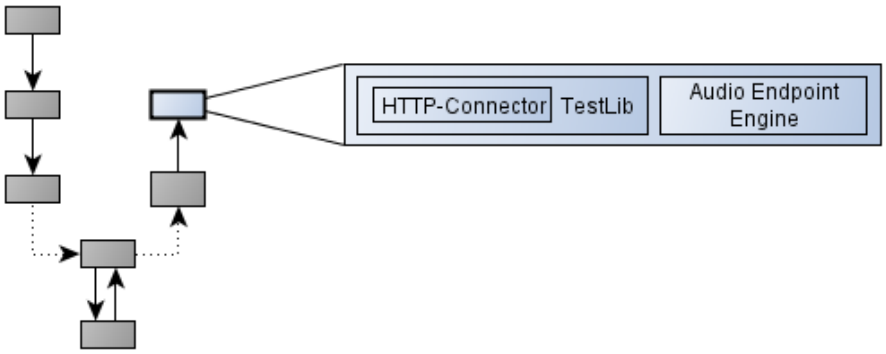


Abbildung 8: Aufbau der Audio-Testsoftware

Ein Teil der TestLib wird durch einen HTTP Connector gebildet, welcher eine rudimentäre Implementierung eines HTTP-Servers bereitstellt. Diese ist in einem solchen Maß generisch gehalten, dass er sich lediglich um die Verwaltung der Verbindung kümmert. Die Verarbeitung der eingehenden Verbindungen und deren Inhalte kann durch externen Code erfolgen, sodass auch an dieser Stelle eine große Flexibilität erreicht wird.

Der HTTP Connector bildet einen guten Ansatzpunkt zum Aufbau der Verbindung zwischen dem TNM und der Testsoftware. Die Protocol Buffer-Nachrichten können als Datenteil von HTTP-Nachrichten gesendet werden. Somit ist seitens der Testsoftware lediglich eine Schicht zur Verarbeitung der eingehenden Verbindungen zu implementieren, und auf Seite des TNMs eine entsprechende HTTP-Gegenstelle, welche die erforderlichen Nachrichten an den HTTP Connector übermitteln kann. Da seitens der Testsoftware jederzeit Ereignisse eintreten können, welche unverzüglich an den NESSEE Server als kontrollierende Komponente übermittelt werden müssen, bedarf es auch eines Rückkanals. Das HTTP-Protokoll sieht allerdings keine Möglichkeit vor, eine Nachricht vom Server zum Client zu übertragen, wenn dieser nicht eine Anfrage vom Client an den Server vorausging. Um dieses Problem zu lösen gibt es verschiedene Möglichkeiten. Zum einen könnte man vom TNM aus den Server der TestLib periodisch nach Ereignissen abfragen. Dieses Vorgehen wird auch „Polling“ genannt. Dabei stellt sich die Frage, in welchem Intervall dies zu tun wäre.

Einerseits möchte man über aufgetretene Ereignisse möglichst zeitnah informiert werden, was für kurze Abfrageintervalle spricht. Kurze Intervalle bedeuten andererseits auch häufige Nachrichten, wodurch die Belastung des Netzwerkes stark ansteigen würde.

Diesem Konflikt kann man begegnen, indem man das sogenannte „Long Polling“ als eine Abwandlung des klassischen Pollings verwendet. Hierbei sendet der Client eine Anfrage nach Ereignissen an den Server. Sind dem Server zum aktuellen Zeitpunkt keine Ereignisse bekannt, über die er den Client informieren kann, wird jedoch keine leere Antwort zurück gesendet, wie es beim klassischen Polling der Fall wäre. Stattdessen wird die Anfrage des Clients offen gehalten. Sobald nun serverseitig ein Ereignis auftritt, über welches der Client zu informieren ist, wird die Antwort auf die noch offene Anfrage erstellt und gesendet. Auf diese Weise wird die sonst entstehende Zeitspanne zwischen dem Auftreten eines Ereignisses und der letzten Anfrage des Clients auf ein vernachlässigbar kleines Minimum reduziert, sodass der Client nahezu in Echtzeit informiert wird. Dennoch ist der Long Polling Ansatz für die geplante Integrationslösung nicht interessant. Einer der Gründe dafür ist, dass HTTP nicht dafür geeignet ist, mehr als eine Anfrage gleichzeitig über die selbe Verbindung zu verarbeiten. Sollen während des Wartens auf eine Ereignisbenachrichtigung noch Befehle an den Server übertragen werden, so muss die offene Anfrage zuerst geschlossen, dann der Befehl übertragen, und abschließend eine neue Long Polling Anfrage gestellt werden. Darüber hinaus sind jede Anfrage und jede Antwort mit einem Header versehen. Dieser ermöglicht zwar die einfache Übertragung von Meta-Informationen wie beispielsweise einem Fehlercode. Gemessen an der Anzahl und der Größe der zu erwartenden Nachrichten würde dieser Header jedoch einen deutlichen Mehraufwand darstellen, da mehr als die Hälfte der definierten Nachrichten während der Übertragung kleiner wäre als der Header.

Eine andere Möglichkeit ist, das TNM ebenfalls mit einem einfachen HTTP-Connector auszustatten. Auf diese Weise kann die Testsoftware bei Auftreten eines Ereignisses von sich aus eine Nachricht an das TNM übermitteln, auf welche es dann angemessen reagieren kann. Diese Lösung erhöht zwar den Aufwand der Implementierung, da die Verbindung zwischen TNM und SUT in zweifacher Ausführung umgesetzt werden muss, nutzt die verfügbaren Ressourcen jedoch nur bei entsprechendem Bedarf und somit wesentlich sparsamer. Die beschriebene Lösung wird durch **Abbildung 9** skizziert.

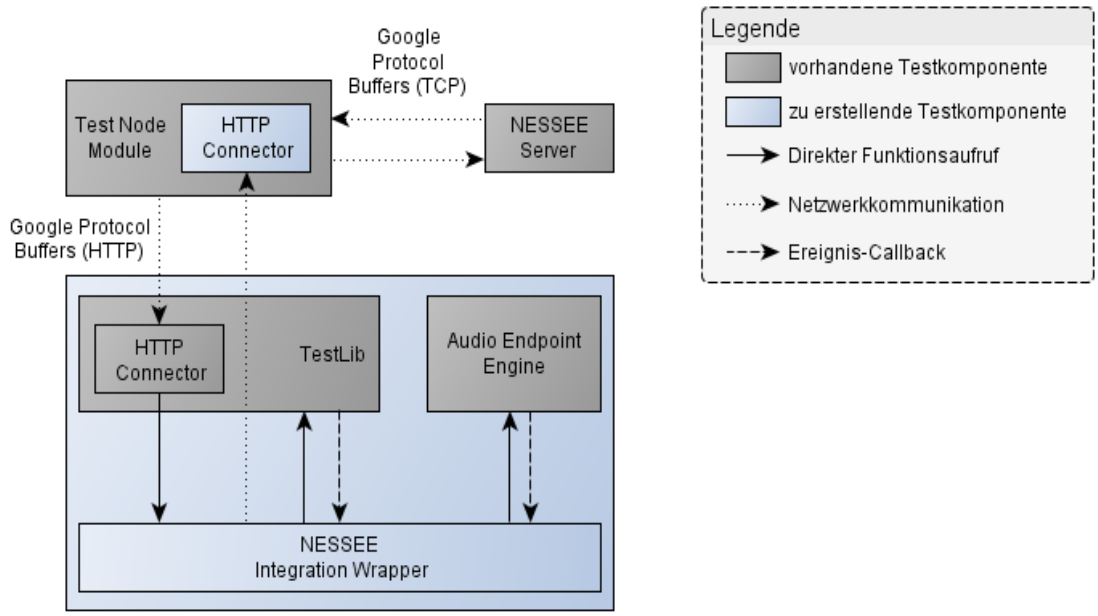


Abbildung 9: Kommunikation zwischen TNM und SUT via HTTP

Es ist jedoch auch ein anderer Ansatz zur Lösung denkbar. Dieser besteht darin, auf die Nutzung des bereits vorhandenen HTTP-Connectors zu verzichten und die Kommunikation zwischen TNM und SUT unter Nutzung einer reinen TCP-Verbindung zu realisieren. Hierfür muss die Kommunikation auf beiden Seiten implementiert werden, was einen gewissen Mehraufwand darstellt. Auf diese Weise kann jedoch der gesamte Nachrichten-Austausch über eine Verbindung erfolgen.

Die Verbindung zwischen Server und TNM über TCP schafft ein offenes und sehr flexibles System, da die Steuerung der TNMs auch auf andere Art als durch den NESSEE Server erfolgen kann. Dieses Maß an Flexibilität und das Prinzip der Wiederverwendbarkeit in der Softwareentwicklung legen nahe, auch den zweiten Teil des Kommunikationsweges über TCP abzubilden. Durch diese Vereinheitlichung würde eine bessere Wartbarkeit erreicht und die Offenheit gefördert werden.

Die bestehende Kommunikation zwischen Server und TNM auf Basis der Protocol Buffers via TCP wird hierbei ideal ergänzt durch die Weiterführung der Kommunikation zwischen TNM und der Testsoftware, ebenfalls auf Basis der Protocol Buffers via TCP. Die Verarbeitung der eingehenden Kommandos erfolgt daraufhin im zu entwickelnden NESSEE Integration Wrapper, welcher an die Stelle des COM-Wrappers bei den Videotests tritt. Der Integration Wrapper kann seinerseits auf Funktionen der TestLib und der Audio Endpoint Engine zugreifen und somit die Endanwendung steuern.

Diese Lösung wird in **Abbildung 10** gezeigt und wird aufgrund ihrer Vorteile der Realisierung über HTTP vorgezogen.

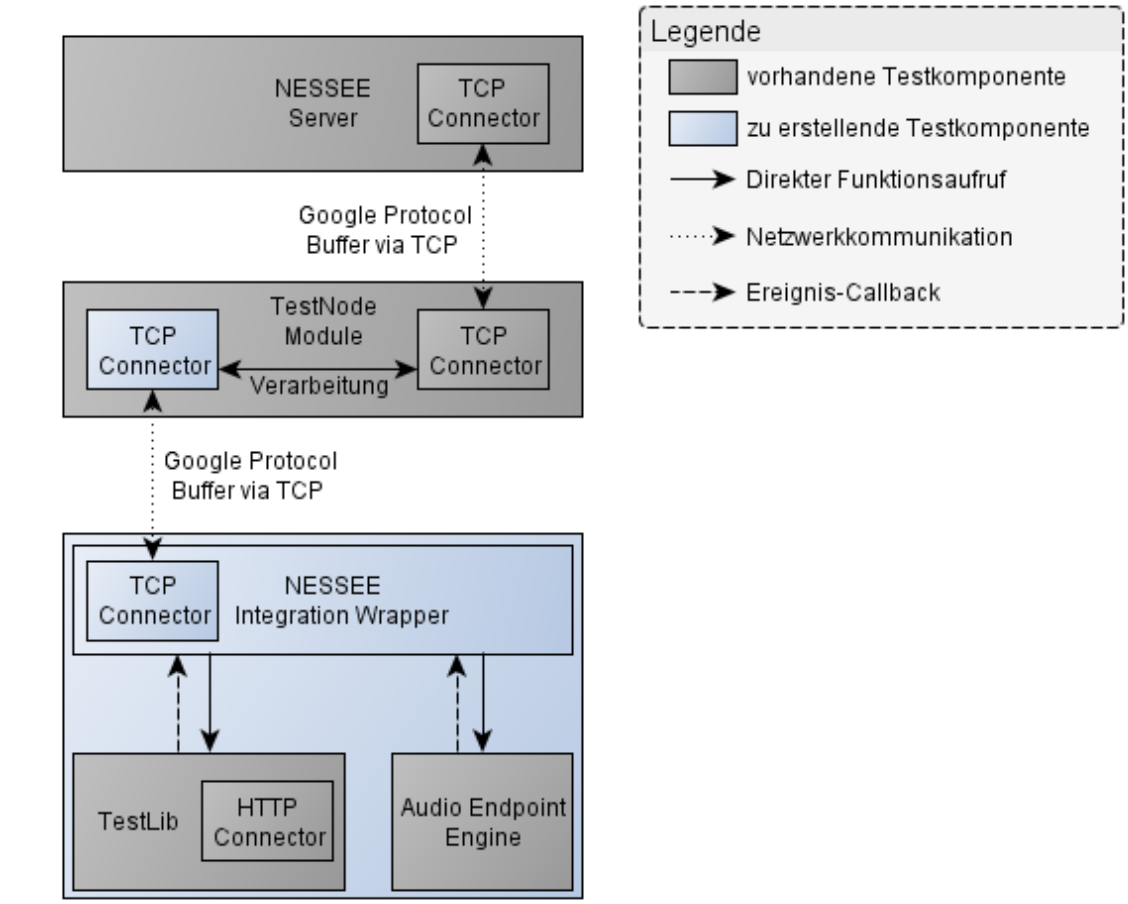


Abbildung 10: Kommunikation zwischen TNM und SUT via TCP

Im Folgenden soll anhand eines konkreten Beispiels der genaue Ablauf während eines laufenden Tests geschildert werden. Dazu wird angenommen, dass der Test bereits gestartet wurde und auf die Anweisung trifft, die Aufnahmelautstärke des Mikrofons aller Testteilnehmer auf 50% festzulegen. Der Ablauf wird durch **Abbildung 11** visualisiert.

Im Beispiel stellen die Server Test Script Libs eine Funktion „SetMicVolForAll“ bereit, welche die Ziellautstärke als Gleitkommazahl zwischen 0 und 1 entgegennimmt und aus dem eigentlichen Testscript aufgerufen wird. Durch eine mithilfe des Microsoft ScriptControls bereitgestellte Umgebungsvariable ist es der Funktion möglich, Funktionen des NESSEE Servers aufzurufen.

In diesem Fall ist es „ExecuteForAllUsersEx“, um die in den Parametern festgelegte Funktion auf Clientseite für alle laufenden Instanzen der Testsoftware auszuführen. „ExecuteForAllUsersEx“ übernimmt zu diesem Zweck den Namen der Zielfunktion, hier „SetMicVolume“, sowie die Parameter dieser Funktion, in diesem Fall den relativen Wert 0,5 für 50%. Alle Parameter werden dabei als Zeichenkette übertragen.

Der NESSEE Server verwaltet die Verbindungen zu allen an diesem Test teilnehmenden TNMs, sodass er im Folgenden jedem der TNMs den Befehl übermitteln kann, die angegebene Funktion durch alle auf dem Computer laufenden Instanzen der Testsoftware ausführen zu lassen. Die Übermittlung dieser Befehle erfolgt asynchron, es wird also nicht auf das Ergebnis der Abarbeitung gewartet, bevor das nächste TNM angesprochen wird. Da die Ausführung des Befehls auf einem TNM nicht vom Abschluss der Abarbeitung der Operation auf dem vorherigen TNM abhängt, lässt sich durch dieses Vorgehen die Gesamtzeit bei der Verarbeitung aller Operationen verkürzen. So ist es möglich, auch bei vielen verbundenen Testsystemen eine möglichst zügige Übermittlung der Befehle zu erreichen. Im Anschluss kehrt der serverseitige Kontrollfluss zu den Testscripten zurück. In diesen wird daraufhin gewartet, bis durch eine Ereignismeldung die vollständige Abarbeitung des Befehls durch alle SUT-Instanzen bestätigt wird.

Jedes TNM kennt seinerseits alle Instanzen der SUT, welche es verwaltet. Somit kann nun der Befehl zum Setzen der Mikrofonlautstärke für jede der laufenden Instanzen an die clientseitige Test Script Lib übergeben werden. An dieser Stelle besteht die Möglichkeit, weitere Aktionen auszuführen oder Eigenschaften zu ändern, bevor die Funktion des TNM aufgerufen wird, welche den Befehl an die SUT übermittelt. Der Zugriff auf die Funktionen des TNM durch die Client Test Script Lib erfolgt auch hier wieder mithilfe des Microsoft Script Controls, welches ein entsprechendes Zugriffsobjekt bereitstellt.

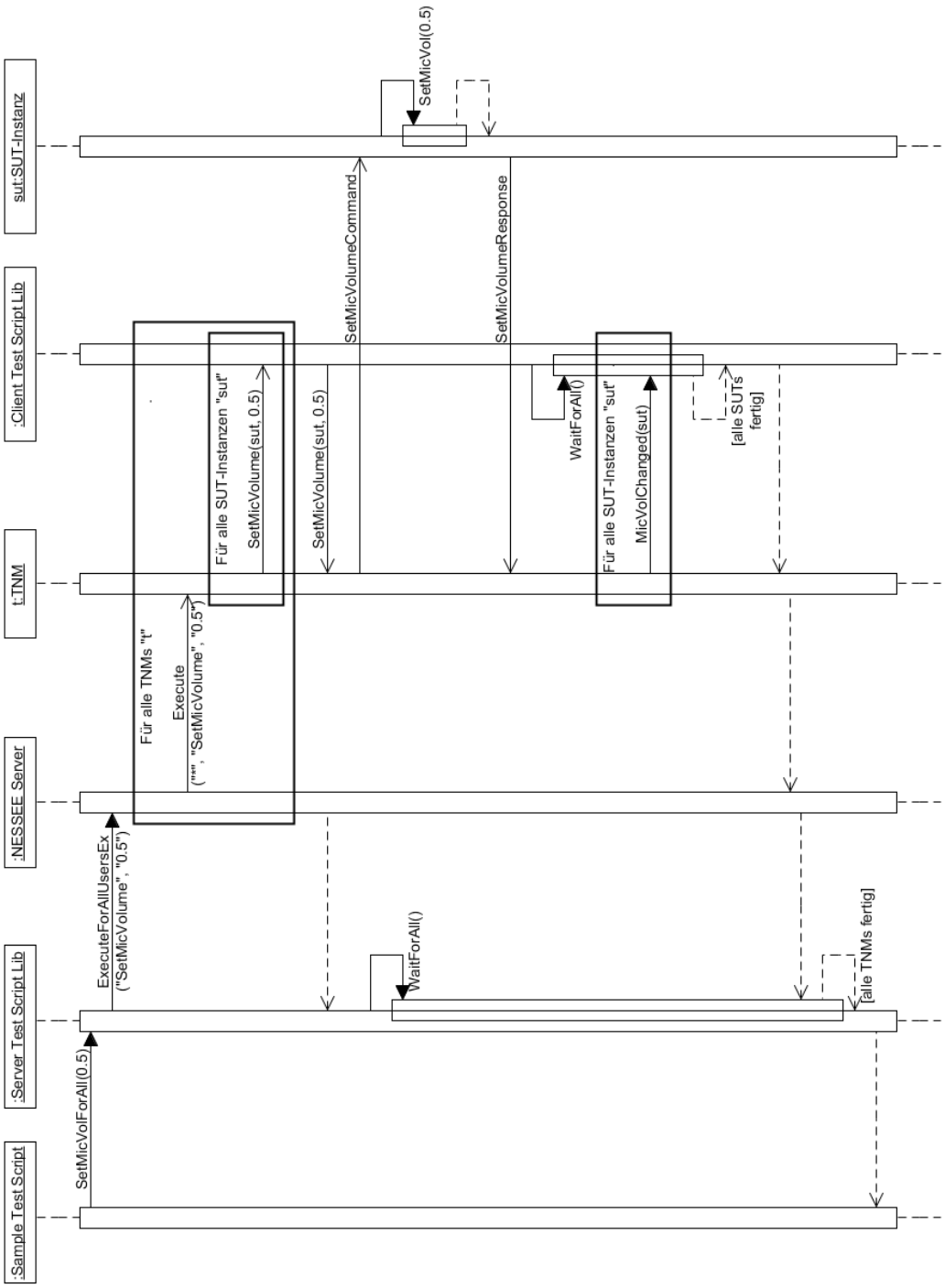


Abbildung 11: Zeitlicher Verlauf einer beispielhaften Testanweisung

Der Befehl wird durch das TNM schließlich an alle Instanzen der SUT übertragen, in welchen er verarbeitet wird. Dies erfolgt durch die Übertragung der entsprechenden Protocol Buffer Nachrichten. In diesem Beispiel handelt es sich konkret um „SetMicVolumeCommand“. Das Ergebnis der Verarbeitung kann in diesem Fall das erfolgreiche Setzen der Lautstärke sein, oder aber ein Misserfolg. Dieser kann verschiedene Gründe haben, beispielsweise könnte das Mikrofon vor dem Empfang des Befehls entfernt worden sein. Das Ergebnis der Ausführung wird im Anschluss an das TNM übermittelt, welches an dieser Stelle eine zuvor festgelegte Methode der clientseitigen Test Script Lib aufruft. Diese Benachrichtigung des TNM erfolgt ebenfalls durch das Senden einer Protocol Buffer Nachricht, hier „SetMicVolumeResponse“. In den Test Script Libs werden unterdessen die Ergebnisse aller verbundenen Instanzen der SUT erwartet. Sobald sie empfangen wurden gibt das TNM alle Ergebnisse und den parallelen, clientseitigen Kontrollfluss zurück an den NESSEE Server, welcher die eingehenden Ergebnisse an die serverseitigen Test Script Libs weitergibt. Die Rückkehr des Kontrollflusses kann auch erfolgen, wenn nach dem Ablauf einer festgelegten Zeitspanne, dem sogenannten Timeout, keine Rückmeldung über das Ergebnis der Abarbeitung durch die SUT erfolgt.

Sind die Ergebnisse aller SUT-Instanzen empfangen und ausgewertet, kann mit der Testausführung fortgefahren werden. An dieser Stelle muss auf eventuelle Misserfolge reagiert werden, um anschließend entscheiden zu können, ob der Test vorzeitig beendet werden muss. Anderenfalls gelangt der Kontrollfluss zurück in das entsprechende Testscript, von wo aus mit der Verarbeitung der nächsten Anweisung begonnen wird. Der beschriebene Ablauf wiederholt sich entsprechend.

4.3 Konkrete Schritte zur Realisierung

4.3.1 Erstellung des NESSEE Integration Wrappers

Umfang und Aufbau des Integration Wrappers orientieren sich maßgeblich an der bereitgestellten Schnittstellenbeschreibung, welche durch ein anderes Team entwickelt wurde. Zum besseren Verständnis ist der Aufbau dieses Interfaces im Anhang enthalten. Die durch das Interface zur Verfügung gestellten Funktionen müssen über das Kommunikations-Protokoll dem TNM zugänglich gemacht werden. Der erste Schritt ist es daher, die entsprechenden Dateien für die Protocol Buffer Nachrichten zu erstellen, welche die übertragbaren Nachrichten und damit das eigentliche Protokoll definieren.

Protocol Buffer Nachrichten werden zumeist unter Verwendung von *.proto-Dateien definiert. Zu deren Aufbau sei an dieser Stelle auf die offizielle Dokumentation durch Google verwiesen ([Google_2]).

Mithilfe des durch Google bereitgestellten Protocol Buffer Compilers[1] werden die erstellten Dateien anschließend in C++-Quelltexte überführt. Das Ergebnis dieses Vorgangs ist eine Sammlung von Klassen, welche in ihrer Struktur genau den Nachrichten entsprechen. Ihre Aufgabe ist später die Erstellung dieser Nachrichten aus dem Quelltext heraus, ihre Serialisierung und die anschließende Deserialisierung auf der Gegenseite. Ziel dieses Schrittes ist es, die gesamte Schnittstellenbeschreibung mithilfe von Protocol Buffer Nachrichten abzubilden. Die entstandenen Dateien sind auszugsweise im Anhang enthalten.

Im Folgenden müssen als Reaktion auf eingehende Nachrichten die vorhandenen Funktionen der Testsoftware in Abhängigkeit vom Typ der Nachricht aufgerufen werden. Auftretende Ereignisse müssen behandelt werden, indem die zum jeweiligen Ereignis passende Nachricht erstellt und an das TNM übertragen wird.

4.3.2 Erweiterung der JavaScript-Bibliothek

Die bestehende JavaScript-Bibliothek teilt sich in zahlreiche Dateien auf, welche die videospezifischen Testfunktionen kapseln. Nach diesem Schema werden zwei weitere Dateien erstellt. Die zur Verwendung im Testscript verfügbaren Funktionen auf Serverseite werden unter „server_audio.js“ gespeichert, die durch das TNM aufrufbaren Funktionen auf Clientseite unter „client_audio.js“. Beide Dateien können in der Datei „Behavior.xml“ referenziert werden, um auf die enthaltenen Funktionsdefinitionen zugreifen zu können. Für jede Funktion, welche durch die neue Testsoftware bereitgestellt wird, ist eine Funktion in der Test Script Lib definiert. Weiterhin gibt es eine Funktion für jedes der definierten Ereignisse, welche bei dessen Auftreten aufgerufen wird.

4.3.3 Erweiterung der TDL

Eine vollständige Testfallbeschreibung sieht drei wesentliche XML-Dateien vor, um die Konfiguration eines Testfalls abzubilden. Zum einen die Datei „Behavior.xml“, welche einerseits alle verfügbaren Ereignisarten und Testzustände definiert, und andererseits alle ausführbaren Aktionen festlegt.

1 Zu finden unter https://code.google.com/p/protobuf/downloads/list

Eine solche Aktion kann beispielsweise der Aufruf einer JavaScript-Funktion sein, ein möglicher Zustand wäre „TestCreated“. Die zweite Datei ist „NetworkTopology.xml“. Sie stellt die Möglichkeit zur Verfügung, die zu emulierende Netzwerkarchitektur festzulegen. Hier können beispielsweise Bandbreite, Paketverlust und andere Einflüsse bestimmt werden. Darüber hinaus ist es möglich, die Gültigkeit einer Einschränkung auf eine Richtung zu begrenzen. So kann zum Beispiel die Verbindung vom Endpunkt zum Server anders geartet sein als die entsprechende Verbindung in Gegenrichtung.

In der Datei „TestCaseDescription.xml“ werden die beiden genannten Dateien schließlich eingebunden. Darüber hinaus wird hier festgelegt, über welche Eigenschaften die jeweiligen Testsysteme verfügen, und wie oft sie vertreten sind. Beispielsweise kann hier konfiguriert werden, dass ein bestimmter Endpunkt einen Videostrom in einer Auflösung von 400x350 Pixeln empfangen möchte, oder in einem Seitenverhältnis von 16:9. In diesem Fall ist die Rede von sogenannten „Video Capabilities“.

Ein entsprechendes Beispiel zu jeder der drei Dateien ist als Auszug im Anhang zu finden. Das Studium dieser Dateien hat ergeben, dass lediglich das Schema der „TestCaseDescription“-Datei angepasst werden muss. Entsprechend der Konfiguration für Videotests müssen an dieser Stelle „Audio Capabilities“ eingefügt werden. Die entsprechenden Eigenschaften eines Audio-Endpunktes sind:

- Welcher Audio-Codec soll zum Senden verwendet werden?
- Soll die Rauschunterdrückung aktiviert werden?
- Soll die Echo-Unterdrückung aktiviert werden?
- Soll der Audiostrom aufgenommen werden, und wenn ja, in welche Datei?
- Wieviele „Performance Points“ benötigt dieser Endpunkt?

Die Performance Points sind hierbei ein Maß für den Bedarf an Rechenleistung der Testanwendung auf dem jeweiligen Testsystem. Sie werden benötigt, da üblicher Weise mehrere dieser Endpunkte auf einem Computer ausgeführt werden. Im Vorfeld wird dazu aus den Eigenschaften des Systems ermittelt, wie viele dieser Performance Points der entsprechende Computer bereit stellen kann. Bei virtuellen Maschinen wird derzeit mit einer festen Größe gearbeitet. Dieses Maß der zur Verfügung stehenden Leistung wird mit jedem gestarteten Endpunkt um dessen benötigte Performance Points reduziert. So kann die Ausführung von Tests stets mit wenigen Computern erfolgen.

Eine Unterscheidung in einen reinen Video- oder Audio-Test ist indes nicht nötig. Aus dem Vorhandensein oder Fehlen der verschiedenen Capabilities ergibt sich automatisch, worum es sich handelt. Sind lediglich Video Capabilities aufgeführt, handelt es sich um einen reinen Videotest. Das selbe gilt entsprechend für Audiotests. Sollten Eigenschaften beider Arten angegeben sein, so muss derzeitig sowohl ein Video- als auch ein Audio-Endpunkt gestartet werden. Geplant ist die Erstellung einer gemeinsamen Video- und Audio-Endanwendung, mit deren Hilfe dann auch kombinierte Tests ausgeführt werden können.

4.3.4 Erweiterung des Protokolls zwischen TNM und NESSEE Server

Das bereits bestehende Protokoll zwischen dem NESSEE Server und dem TNM wird in seiner derzeitigen Implementierung für ausreichend erachtet, um alle neuen Funktionen abbilden zu können. Der Grund dafür liegt darin, dass über dieses Protokoll keine direkten Funktionsaufrufe übergeben werden. Vielmehr verfügt es über generische Befehle wie beispielsweise „ExecuteForOneUser“ oder „ExecuteForAllUsers“. Diese Befehle nehmen den Namen der Zielfunktion sowie deren Parameter als Zeichenketten entgegen. Dies ermöglicht die Angabe jeder beliebigen Funktion zur Ausführung auf Clientseite. Wahlweise kann außerdem der Name einer Funktion angegeben werden, welche serverseitig ausgeführt werden soll, sobald der Aufruf der angegebenen Clientfunktion zurückkehrt. Somit ist auch das Entwickeln von ereignisbasierten Testfallbeschreibungen möglich, sodass an dieser Stelle keinerlei Änderungen vorgenommen werden müssen.

4.3.5 Erweiterung des TNM zur Kommunikation mit der SUT

Die erstellten *.proto-Dateien bilden die Definition des von TNM und NESSEE Integration Wrapper gemeinsam genutzten Protokolls. Damit die Kommunikation problemlos verläuft, müssen beide Seiten dieses Protokoll verstehen und verwenden können. Um dem TNM auf Basis von C# .Net die Verwendung dieses Protokolls zu ermöglichen, müssen die definierten Protocol Buffer Nachrichten für ihre Verwendung in C#-Code überführt werden. Auch für diesen Zweck gibt es ein Werkzeug. Die Bibliothek „protobuf-net“ von Marc Gravell[2] verfügt über einen Parser, welcher in der Lage ist, *.proto-Dateien in C#-Code zu überführen. Somit ist es problemlos möglich, das definierte Protokoll in beiden Sprachen verfügbar zu machen. Ein Aufruf der entsprechenden Funktion im C#-Code des TNM führt somit zu einer Umwandlung in eine Protocol Buffer Nachricht (Serialisierung),

2 Zu finden unter https://code.google.com/p/protobuf-net/

deren Übertragung zum NESSEE Integration Wrapper, und die Rücküberführung in einen Funktionsaufruf im C++-Code des Wrappers (Deserialisierung). Ebenso kann ein auftretendes Ereignis im Code der Testanwendung zurück zum TNM übertragen werden, um letztlich den Server über den aktuellen Zustand der Software zu informieren.

Eine besondere Anforderung liegt hierbei darin, dass der Netzwerk-Datenstrom der Verbindung zwischen TNM und SUT über keine festgelegte Struktur verfügt. Somit ist die Grenze zwischen aufeinander folgenden Nachrichten nicht kennzeichnet. Um die Nachrichten auf der Gegenseite fehlerfrei deserialisieren zu können, muss also eine Möglichkeit gefunden werden, eine Nachricht von einer darauf folgenden zu unterscheiden. Zu diesem Zweck wurde eine Nachricht definiert, welche als Container für alle anderen Nachrichten fungiert (siehe Anhang, „Messaging.proto“). In dieser Nachricht wird für jede definierte Nachricht ein Feld vom Typ der zu übertragenden Nachricht vorgesehen, welches als optional gekennzeichnet wird. Soll nun beispielsweise eine „JoinMeetingCommand“-Nachricht versendet werden, so wird zuerst eine „AudioEndpointMessageContainer“-Nachricht erstellt, deren optionales „joinMeetingCommand“-Feld anschließend mit der bereits erstellten Nachricht des gleichnamigen Typs belegt wird. Die anderen Felder bleiben leer. Erhält nun der Empfänger eine Nachricht, ist durch die getroffene Festlegung bekannt, dass es sich um eine „AudioEndpointMessageContainer“-Nachricht handelt. Ist diese vollständig empfangen, kann überprüft werden, welches der optionalen Felder einen Wert enthält. Durch den Datentyp dieses Feldes wird festgelegt, um was für einen Typ Nachricht es sich handelt. Somit ist die fehlerfreie Weiterverarbeitung der Nutzdaten möglich.

Die Verwendung dieser Container-Nachricht als Träger der eigentlichen Nutzdaten-Nachricht ermöglicht die fehlerfreie Übertragung und Verwendung aller Nachrichten ohne den Aufwand, einen Header mit Meta-Informationen über den Inhalt übertragen zu müssen. Weiterhin wird die Container-Nachricht ebenso serialisiert und komprimiert übertragen wie auch die Nutzdaten. Somit wird die Belastung des Netzwerkes so gering wie möglich gehalten, was den angestrebten hohen Nachrichtendurchsatz ermöglicht.

4.3.6 Erweiterung des Software-Verteilungsprozesses

Da die Testsoftware nicht automatisch auf den genutzten Testsystemen vorhanden sind, bedarf es eines Prozesses, welcher die Software auf das System transportieren kann. Für die Video-Testsoftware gibt es bereits eine entsprechende Möglichkeit, sodass die vorhandene Lösung ebenfalls dazu benutzt werden kann, die Audio-Testsoftware zu transportieren.

Dazu wird die Weboberfläche, welche zur Steuerung des NESSEE Servers benutzt wird, dazu verwendet, eine Kopie der Testsoftware aus einem lokalen Verzeichnis hoch zu laden und an ausgewählte Testsysteme zu übertragen. Auf diese Weise ist auch da einfache Austauschen der Software gegen eine andere Version unkompliziert.

5 Zusammenfassung

Im Rahmen der vorliegenden Arbeit wurde ein Konzept erstellt, auf welche Weise das NESSEE-Testsystem um eine Möglichkeit erweitert werden kann, Audiokomponenten zu testen. Basierend auf dem bestehenden System zum Testen von Videokomponenten wurde dabei untersucht, inwieweit die vorhandene Funktionalität beibehalten werden kann, und an welcher Stelle sie erweitert oder angepasst werden muss.

Auf Grundlage dieses Konzeptes wurde ein Prototyp entwickelt, welcher die Machbarkeit des erstellten Konzeptes unter Beweis stellt. Dazu wurde das vorhandene System der Testfallbeschreibung mittels XML- und JavaScript-Dateien erweitert. Es bietet nun Funktionen an, welche für die Ausführung von Audiotests erforderlich sind. Die neu definierten Funktionen stehen nun zur Verwendung während der Testausführung zu Verfügung. Die bestehende Kommunikation wird benutzt, um einen Funktionsaufruf vom NESSEE Server an die TestNodeModule zu delegieren. Diese wurden um die entsprechende Verarbeitung der neuen Befehle erweitert und können sie ihrerseits an die Testanwendung delegieren. Eine vorhandene Testsoftware wurde dazu um eine netzwerkbasierte Schnittstelle zur Kommunikation erweitert. Das TestNodeModule nutzt diese Schnittstelle zur Übermittlung von Befehlen an die Testsoftware. Definiert werden diese Befehle durch das Protokoll, welches zwischen dem TestNodeModule und der Testsoftware vereinbart wurde. Die erstellte Schnittstelle zur Kommunikation übernimmt letzten Endes die Ausführung der bereit gestellten Funktionen der Testsoftware. Während der Testausführung auftretende Ereignisse werden über den selben Kommunikationskanal an den NESSEE Server zurück übermittelt. Auf diese Weise besteht die Möglichkeit, in den jeweiligen Testscripten flexibel auf diese Ereignisse zu reagieren.

Die Machbarkeit des erarbeiteten Konzeptes wurde mithilfe eines beispielhaften Testscriptes nachgewiesen. Inhalt dieses Scriptes ist die Anforderung zur softwaretechnischen Erstellung eines Audio-Endpunktes und das Erhalten des Ereignisses zur Bestätigung dieser Erstellung. Anschließend soll die erstellte Instanz des Endpunktes wieder gelöscht, sowie ebenfalls das bestätigende Ereignis empfangen werden. **Abbildung 12** zeigt die Ausgaben einer Ausführung dieses Testscriptes.

Die Erweiterung des Software-Verteilungsprozesses war gemäß der Erwartungen unkompliziert umzusetzen und funktioniert ebenso wie geplant.

Abbildung 12: Beispielhafte Ausgabe eines Testlaufes

6 Ausblick

Die prototypische Implementierung hat grundlegend die Machbarkeit des erarbeiteten Konzeptes aufgezeigt. Um die Umsetzung der Integrationslösung zu vervollständigen, sind weitere Arbeiten notwendig, welche an dieser Stelle kurz umrissen werden sollen.

Die Verwendung der aktuellen Implementierung der Testsoftware hat ergeben, dass es Probleme bei der Verwendung auf einem virtuellen Computer gibt. In diesem Fall schlägt aufgrund fehlender Audio-Hardware wie Mikrofon und Lautsprecher bereits die softwaretechnische Erstellung eines Audio-Endpunktes fehlt. Da dieser jedoch für alle weiteren Funktionen essentiell ist, wird durch dieses Verhalten die angestrebte Verwendung auf überwiegend virtualisierter Hardware verhindert. Dieses Problem kann behoben werden, indem Alternativen zu den tatsächlichen Hardwaregeräten angeboten werden. Beispielsweise ist denkbar, eine Aufnahme aus einer Audiodatei statt einem Mikrofon zu verwenden. Zu diesem Zweck bedarf es einer Möglichkeit, die entsprechenden Audiodateien ausgehend vom Server auf ausgewählte Endpunkte zu verteilen. Genauso kann ein wiederzugebender Audiostrom in eine Datei umgeleitet werden, anstatt über Lautsprecher ausgegeben zu werden. In diesem Fall ist dafür Sorge zu tragen, dass die entstandenen Audiodateien nach einem Test von den Endpunkten zurück zum Server übertragen werden können. Auch eine weitgehend automatisierte Möglichkeit der Analyse dieser Dateien hinsichtlich festgelegter Qualitätskriterien wie Echo und Rauschen sollte untersucht werden.

Zum bisherigen Zeitpunkt fehlt auch eine Möglichkeit, einen aktiven Endpunkt zu überwachen. Beim Starten der Testanwendung können Fehler auftreten, ebenso kann der Prozess im Testverlauf abstürzen oder während der Ausführung feststecken. Ein Verfahren, um die Verfügbarkeit der Testanwendung zu überprüfen und dem Server mitzuteilen, ist also unverzichtbar.

Ebenso wurde die Fehlerbehandlung nur soweit umgesetzt, wie es zum Ausführen des Prototypen notwendig war. Alle weiteren Fehler, wie zum Beispiel der Verlust der Verbindung zwischen TestNodeModule und Software Under Test während der Testausführung, müssen erkannt und behandelt werden.

7 Quellenverzeichnis

Kürzel	Quelle	Stand
[Google_1]	https://developers.google.com/protocol-buffers/	16.09.2013
[Google_2]	https://developers.google.com/protocol-buffers/docs/overview	27.09.2013
[ItWissen_1]	http://www.itwissen.info/definition/lexikon/Framework-framework.html	04.11.2013
[ItWissen_2]	http://www.itwissen.info/definition/lexikon/interprocess-communication-IPC.html	04.11.2013
[ItWissen_3]	http://www.itwissen.info/definition/lexikon/logfile-Log-Datei.html	04.11.2013
[ItWissen_4]	http://www.itwissen.info/definition/lexikon/Metadaten-meta-data.html	04.11.2013
[ItWissen_5]	http://www.itwissen.info/definition/lexikon/Router-RO-router.html	04.11.2013
[ItWissen_6]	http://www.itwissen.info/definition/lexikon/Subnetz-SN-subnet.html	04.11.2013
[ItWissen_7]	http://www.itwissen.info/definition/lexikon/Wrapper-wrapper.html	04.11.2013
[Micro_1]	http://www.microsoft.com/de-de/download/details.aspx?id=1949	16.09.2013
[Micro_2]	http://msdn.microsoft.com/en-us/library/aa288468.aspx	16.09.2013
[Micro_3]	http://msdn.microsoft.com/de-de/library/ms680573.aspx	18.09.2013

Alle oben aufgeführten Quellen sind auch als PDF-Druck vom Zeitpunkt des Bezuges auf dem beiliegenden Datenträger verfügbar.

8 Literaturverzeichnis

[DUSTIN u.a. 2001] DUSTIN, E., RASHKA, J., PAUL, J.: Software automatisch testen, Auflage 2001, Springer-Verlag, Heidelberg, 2001

[KÜHNEL 2010] KÜHNEL, A.: Visual C# 2010 – Das umfassende Handbuch, 5. aktualisierte und erweiterte Auflage, Galileo Press, Bonn, 2010

[LÜBKE u.a. 2012] LÜBKE, R., LUNGWITZ, R., SCHUSTER, D:, SCHILL, A.: Emulation of Complex Network Infrastructures for Large-Scale Testing of Distributed Systems, IADIS WWW/Internet 2012 Conference, Madird, Spain; 2012

[LÜBKE u.a. 2012] LÜBKE, R., LUNGWITZ, R., SCHUSTER, D:, SCHILL, A.: Large-Scale Tests of Distributed Systems with Integrated Emulation of Advanced Network Behavior, IADIS International Journal on WWW/Internet, Vol. 10, No. 2, S. 138-151; 2013

[POHL 2008] POHL, K.: Requirements Engineering – Grundlagen, Prinzipien, Techniken,
2., korrigierte Auflage, dpunkt.verlag GmbH, Heidelberg, 2008

[POHL 2011] POHL, K., RUPP, C.: Basiswissen Requirements Engineering – Aus- und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering Foundation Level, 2., überarbeitete Auflage, dpunkt.verlag GmbH, Heidelberg, 2011

[STROUSTRUP 2010] STROUSTRUP, B.: Einführung in die Programmierung mit C++, 1. Auflage, Pearson Studium Verlag, München, 2010

Anhang

Beispielhafte Auszüge der TDL-Dateien

Behavior.xml

```
<!-- import the test case specific javascript files -->
<ScriptImport file="server_main.js" node="coordination"/>
<ScriptImport file="client_main.js" node="client"/>

<!-- Define all EventTypes that are triggered and consumed during the test -->
<SupportedEventTypes>
      <EventType>StatusEvent</EventType>
      <EventType>StatisticsEvent</EventType>
      <EventType>LoginEvent</EventType>
      <EventType>CallEvent</EventType>
      <EventType>HangupEvent</EventType>
      <EventType>LogoutEvent</EventType>
      <EventType>ScriptErrorEvent</EventType>
</SupportedEventTypes>

<!-- Define all TestStates that are supported by this Test -->
<SupportedTestStates>
      <TestState>TestCreated</TestState>
      <TestState>TestStateAllLoggedIn</TestState>
      <TestState>TestStateA</TestState>
      <TestState>TestStateB</TestState>
      <TestState>TestStateC</TestState>
      <TestState>TestStateD</TestState>
</SupportedTestStates>

<!-- Define the Actions that can be triggered during the test -->
<Action id="Auto start">
      <ExecuteScriptFunction node="coordination"
            function="autostart" returns="undefined">
            <TerminalEvent>AutostartTerminal</TerminalEvent>
            <AvailableInTestState>TestCreated</AvailableInTestState>
      </ExecuteScriptFunction>
</Action>
```

NetworkTopology.xml

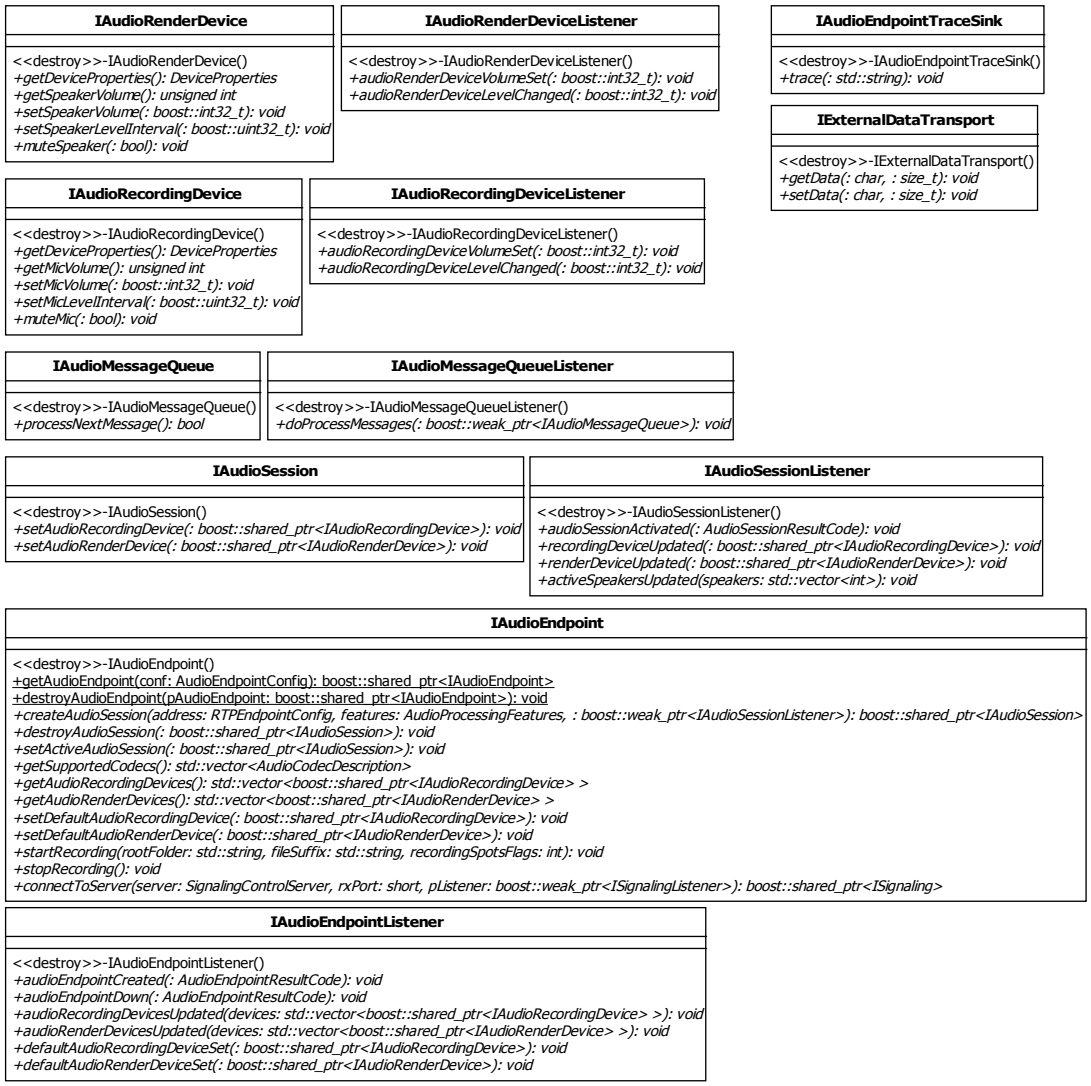
```
<Capabilities>
<!-- Define the different NetworkCapabilities that should be available -->
      <NetworkCapabilities id="LeasedLine">
            <bandwidth direction="both">100000</bandwidth>
            <delay direction="both">15</delay>
      </NetworkCapabilities>

      <NetworkCapabilities id="DSL">
            <loss direction="both">0.1</loss>
            <delay direction="both">12</delay>
            <reordering direction="both">0.2</reordering>
      </NetworkCapabilities>

      <NetworkCapabilities id="DSL2000" BasedOn="DSL">
            <bandwidth direction="down">2000</bandwidth>
            <bandwidth direction="up">500</bandwidth>
      </NetworkCapabilities>

      <NetworkCapabilities id="DSL6000" BasedOn="DSL">
            <bandwidth direction="down">6000</bandwidth>
            <bandwidth direction="up">1000</bandwidth>
      </NetworkCapabilities>
</Capabilties>
```

TestCaseDescription.xml

```
<!-- Import the external modules you want to use in the test case -->
<NetworkTopology src="NetworkTopology.xml"/>
<Behavior src="Behavior.xml"/>

<!-- Create a scenario by defining the conferences and its Participants -->
<AgentCapabilities>
      <VideoCapabilities id="OfficePC">
            <MaxResolutionTx>
                  <AspectRatio>16:9</AspectRatio>
                  <Width>600</Width>
            </MaxResolutionTx>
            <MaxResolutionRx>
                  <Width>400</Width>
                  <Height>350</Height>
            </MaxResolutionRx>
            <PerformancePoints>140</PerformancePoints>
      </VideoCapabilities>

      <VideoCapabilities id="OfficePC-Sending" BasedOn="OfficePC">
            <MaxResolutionRx>
                  <AspectRatio>4:3</AspectRatio>
                  <Width>185</Width>
            </MaxResolutionRx>
            <SendsVideo>true</SendsVideo>
            <PerformancePoints>180</PerformancePoints>
      </VideoCapabilities>
</AgentCapabilities>

<Conference id="testConf1">
<!-- Which VideoCapabilitiescluster should be used for this conference? -->
      <ServerAgentId>serverAgent01</ServerAgentId>
<!-- Which template is the conference based on -->
      <EndpointAgents id="endPointAgent01">
            <EndpointType>VcEngine</EndpointType>
            <NetworkEndpointTypeId>Corp01EndpointB</NetworkEndpointTypeId>
            <CapabilitiesId>OfficePC-Sending</CapabilitiesId>
<!-- How many agents with this configuration should be created -->
            <Number clone="path">2</Number>
      </EndpointAgents>
<!-- How many instances of this conference should be created -->
      <number>2</number>
</Conference>

<ServerAgent id="serverAgent01">
      <ServerType>VideoServer</ServerType>
      <NetworkServerId>remote:vidCluster01</NetworkServerId>
      <CapabilitiesId>OfficePC</CapabilitiesId>
</ServerAgent>
```

Struktur der bereitgestellten Schnittstellenbeschreibung

Die folgende Abbildung wurde unter Zuhilfenahme der Software „StarUML“ aus den bereitgestellten Programmdateien automatisch generiert.



Auszüge aus den erstellten *.proto-Dateien

AudioEndpointTypes.proto

```
package citrix.rtc.audioendpoint.Messaging;

enum eAudioCodec
{
	eAudioCodecUnknown = 0;
	eAudioCodecISAC = 1;
	eAudioCodecPCMU = 2;
	eAudioCodecPCMA = 3;
	eAudioCodecILBC = 4;
	eAudioCodecG7220 = 5;
	eAudioCodecG7221_16 = 6;
	eAudioCodecG7221_24 = 7;
	eAudioCodecG7221_32 = 8;
}

message tAudioCodecDescription
{
	required eAudioCodec codecId = 1;
	required string name = 2;
	required int32 sampleFrequency = 3;
	required int32 samplesPerFrame = 4;
	required int32 audioChannels = 5;
	required int32 bitRate = 6;
}

message tRTPEndpointConfig
{
	required string ip = 1;
	required uint32 port = 2;
	required uint32 localPort = 3;
	required int32 rtpPayloadType = 4;
	required tAudioCodecDescription sendCodec = 5;
}
...
```

Signaling.proto

```
import "AudioEndpointTypes.proto";

package citrix.rtc.audioendpoint.Messaging;

message tSignalingControlServer
{
	required string ip = 1;
	required uint32 port = 2;
}

message tSignalingJoinInfo
{
	required string meetingId = 1;
	required string username = 2;
	required tRTPEndpointConfig localEndpoint = 3;
}

message JoinMeetingCommand
{
	required tSignalingJoinInfo connectionInfo = 1;
}

...
```

AudioEnpoint.proto

```
import "Signaling.proto";
import "AudioEndpointTypes.proto";

package citrix.rtc.audioendpoint.Messaging;

////////////////////////////////////////////
// Audio Endpoint
//
message CreateAudioSessionCommand
{
	required AudioEndpointTypes.tRTPEndpointConfig address = 1;
	required AudioEndpointTypes.tAudioProcessingFeatures features = 2;
	required uint32 AudioSessionListener = 3;
}

...
```

Messaging.proto

```
import "AudioEndpointTypes.proto";
import "Signaling.proto";
import "AudioEndpoint.proto";

package citrix.rtc.audioendpoint.Messaging;

message AudioEndpointMessageContainer
{
 //
 // Signaling
 //
 optional JoinMeetingCommand joinMeetingCommand = 1;
 optional Signaling_StartRecordingCommand signaling_StartRecordingCommand = 2;
 optional Signaling_StopRecordingCommand signaling_StopRecordingCommand = 3;
 optional SetRecordingInformationCommand setRecordingInformationCommand = 4;
 optional StartPlaybackCommand startPlaybackCommand = 5;
 optional StopPlaybackCommand stopPlaybackCommand = 6;
 optional SetMuteStateCommand setMuteStateCommand = 7;
 optional SendMessageCommand sendMessageCommand = 8;
 optional DisconnectUserCommand disconnectUserCommand = 9;

 optional JoinResultEvent joinResultEvent = 10;
 optional ConnectedEvent connectedEvent = 11;
 optional DisconnectedEvent disconnectedEvent = 12;
 optional RecordingStateUpdatedEvent recordingStateUpdatedEvent = 13;
 optional PlaybackStateUpdatedEvent playbackStateUpdatedEvent = 14;
 optional MuteStateUpdatedEvent muteStateUpdatedEvent = 15;
 optional UserListUpdatedEvent userListUpdatedEvent = 16;
 optional MessageReceivedEvent messageReceivedEvent = 17;

 //
 // AudioEndpoint
 //
 optional GetAudioEndpointCommand getAudioEndpointCommand = 18;
 optional GetAudioEndpointResponse getAudioEndpointResponse = 19;
 optional DestroyAudioEndpointCommand destroyAudioEndpointCommand = 20;
 optional CreateAudioSessionCommand createAudioSessionCommand = 21;
 optional DestroyAudioSessionCommand destroyAudioSessionCommand = 22;
 optional SetActiveAudioSessionCommand setActiveAudioSessionCommand = 23;
...
```

Selbstständigkeitserklärung

Ich versichere durch meine Unterschrift, dass ich die vorstehende Bachelorarbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften oder dem Internet entnommen worden sind, sind als solche kenntlich gemacht. Keine weiteren Personen waren an der geistigen Herstellung der vorliegenden Arbeit beteiligt. Die Arbeit hat noch nicht in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung dieser oder einer anderen Prüfungsinstanz vorgelegen.

_____ _____

Ort, Datum Unterschrift