

Dokumentation VanadiumCast

Silas Della Contrada

12/18/2020

Inhaltsverzeichnis

1	Vorwort	2
1.1	Hinweis zur Formatierung	2
2	Projektidee	3
3	Motivation	3
4	Eine kurze Einführung in AV-Verarbeitung	4
4.1	Videocodecs	4
4.2	Audiocodecs	4
4.3	Containerformate	4
5	Umsetzung	5
5.1	Bibliotheken und Frameworks	5
5.1.1	Qt	5
5.1.2	QtAV	6
5.2	Entwicklungsverlauf	7
5.2.1	Phase 1: Anforderungen festlegen	7
5.2.2	Phase 2: Entwicklung	7
5.2.3	Phase 3: Optimierungen und Tests	7
5.2.4	Phase 4: Veröffentlichung	8
6	Architektur	9
6.1	Komponenten	9
6.1.1	GUI (QML)	9
6.1.2	API (C++)	11
6.1.3	Devices (C++)	11
6.1.4	Streaming (C++)	11
6.1.5	Sink (C++)	12

1 Vorwort

Google Chromecast ist weit verbreitet, allerdings ist es kein Open-Source und man ist gezwungen ein Google Chromecast oder ein Android TV als Ziel und entweder Chrome oder ein Android-Gerät als Quelle zu benutzen. Außerdem gibt es keine Möglichkeit den Desktop zu übertragen, wodurch eine potenziell gute Lösung eine große Nutzergruppe nicht anspricht. In diese Lücke fügt sich nun VanadiumCast ein.

1.1 Hinweis zur Formatierung

Verweise zu Quellen und weiterführender Literatur sind mit einer dreistelligen Buchstabenkombination in eckigen Klammern dargestellt:

Bla bla bla... [ExW]

bla bla bla... [ExB]

Die Einträge im Literaturverzeichnis lauten dann wie folgt:

[ExW] *Example website*: <https://example.com/examplesite.html> [2020-12-18]

[ExB] Max Mustermann (2020), *Example book*, Beispielverlag, ISBN: 1234567890

2 Projektidee

Mit VanadiumCast kann man Videos und (in Zukunft) seinen Desktop an ein beliebiges Gerät übertragen, auf dem das Programm installiert ist. Es ist ähnlich einfach zu bedienen wie Chromecast, sodass es auch für technisch unbedarfte Personen nutzbar ist. Lediglich die Installation der Endgeräte ist deutlich schwieriger, da diese aber nur einmalig erfolgen muss, gibt es dort keine Probleme.

Zudem ist geplant, dass VanadiumCast auf allen Desktop- und Mobilgeräten installiert und benutzt werden kann, sodass auch Handy→Handy oder Desktop→Handy Übertragungen möglich sind. Als Zielgerät eignet sich aktuell nur Laptops / PCs / Mini-PCs / SBCs mit Intel-CPU's mit iGPU's ab der 3. Generation oder Nvidia-GPU's ab der GTX 8xx, als Quellgerät die gleichen Gerätetypen, jedoch werden hier Intel-CPU's ab der 4. Generation vorausgesetzt, da auf der Quelle mehr Leistung benötigt wird als auf dem Ziel. Es ist aber geplant beide Seiten zu optimieren und zu erweitern, sodass auch kleine SBCs wie ein Raspberry Pi oder Pine A64 als Ziel und/oder Quelle geeignet sind.

Damit ist es dann auch möglich, ganze Schulen mit günstigen SBCs auszustatten, um Schülerinnen und Schülern sowie Lehrkräften Bildschirmübertragungen zu ermöglichen, ohne den komplizierten Weg über Miracast zu gehen, der nur an Fernsehern und in Kombination mit Android oder Windows 10 möglich ist. Da die Software nur im LAN kommuniziert, ist auch der Datenschutz gewährleistet, bei Google Chromecast werden hingegen Daten in die Google Cloud gesendet. Zudem lässt sich bei VanadiumCast die Übertragungsqualität einstellen, sodass man nicht darauf angewiesen ist, dass die Software die richtige auswählt.

3 Motivation

Weil ich persönlich keinen Google Chromecast kaufen wollte und man damit auch keine Videos von Linux aus streamen könnte, habe ich mir am Ende der Sommerferien 2020 überlegt, ob man nicht auch selbst eine Lösung dafür entwickeln könnte. Also habe ich mich im Internet über mögliche C++-Bibliotheken informiert und es wurde schnell klar, dass die sehr einfache Idee recht schwierig umzusetzen ist. Da ich zu diesem Zeitpunkt aber an keinen anderen Projekte gearbeitet habe, nahm ich die Herausforderung an. Das gesamte Projekt ist ein "learning-by-doing"-Projekt, das heißt, dass ich mich dann über Dinge informiere, wenn ich sie konkret benötige. Zum Beispiel habe ich mich zuerst mit FFmpeg bzw. den Libav-Bibliotheken auseinandergesetzt, um zu verstehen, wie man ein Video in C++ transkodieren kann.

4 Eine kurze Einführung in AV-Verarbeitung

4.1 Videocodecs

Videos werden nicht im Rohformat abgespeichert, da sie unkomprimiert sehr viel Platz verbrauchen. Für einen Spielfilm auf Bluray, d.h. 120 min. Full HD mit 1 Byte pro Farbe und 24 Bildern pro Sekunde, sind das bereits:

$$1920 * 1080 * 3B * 24 * 120 * 60s = 1,074,954,240,000 B \approx 1,1 TB$$

Diese Menge an Speicherplatz möchte niemand für einen Film auf seiner Festplatte bereitstellen. Daher wurde 1988 der Kompressionsstandard H.261 festgelegt, dessen Nachfolger zum Veröffentlichungszeitpunkt immer den aktuellen Stand der Technik darstellten, aktuell sind das H.264/AVC und H.265/HEVC, welche auch in VanadiumCast zum Einsatz kommen. In GPUs stecken seit ca. 10 Jahren Videoprozessoren, die die CPU beim En-/Dekodieren von Videos entlasten, sodass weniger Energie verbraucht wird. Erst dadurch wird es möglich, auf Handys, Tablets etc. Video abzuspielen, da die Akkulaufzeit sonst zu gering wäre.

4.2 Audiocodecs

Bei Audiodaten ist das Problem recht ähnlich, und auch wenn man auf aktuellen SD-Karten mehrere Stunden unkomprimiertes Audio speichern kann, ist es meistens ausreichend und teilweise besser, Audio zu komprimieren, z.B. bei Streaming-Diensten wie Spotify, da mobile Daten meist teuer und limitiert sind. Ursprünglich gab es das Problem, dass man auf Walkmans nicht ausreichend Platz hatte, um mehrere Stunden Musik zu speichern, weswegen eine Arbeitsgruppe des Fraunhofer IIS in Zusammenarbeit mit einer anderen Universität, AT&T Bell Labs und Thomson die MP3-Kodierung entwickelte. Dabei wird Platz gespart, indem durch die Anwendung von Psychoakustik gezielt Teile der Daten weggelassen oder verkleinert werden, die der Mensch normalerweise nicht wahrnimmt. Dadurch wird bei einer Bitrate von 192 kBit/s eine Kompressionsrate von 85% gegenüber einer Audio-CD erzielt. Aktuellere Codecs wie AAC und Opus unterscheiden sich von MP3 im Kompressionsalgorithmus und der Qualität bei bestimmten Bitraten, sodass sie unterschiedlich gut für das Streaming geeignet sind. In VanadiumCast wird aktuell AAC verwendet, da es die gleiche Qualität wie MP3 bei geringerer Bitrate gewährleistet.

4.3 Containerformate

Ein Video- oder Audiocodec alleine reicht nicht aus, um eine Video mit Ton und eventuellen Untertiteln etc. abzuspeichern, daher gibt es Containerformate, durch die mehrere Video-/Audiospuren teilweise mit Untertiteln in einer Datei vereint werden. Die Auswahl des Containerformat nachezu keinen Unterschied bei der Dateigröße, sondern nur in den Möglichkeiten, welche Art von Daten enthalten sein können. Bekannte Beispiele sind das von Apple entwickelte und weit verbreitete MP4 oder das offene Matroska-Format, ersteres unterstützt nur Video- und Audiospuren, letzteres alle möglichen Spuren inklusive 3D-Inhalte und Menüs wie auf DVDs.

5 Umsetzung

5.1 Bibliotheken und Frameworks

Ursprünglich habe ich mich direkt mit FFmpeg und den Libav-Bibliotheken auseinandergesetzt, um den Transcoder umzusetzen, aber der Code des Transcoders wurde mit steigender Komplexität des Projekts nicht mehr verständlich, weswegen ich mich nach einer Alternative umgeschaut habe. Dabei bin ich auf das Qt-Plugin QtAV gestoßen, welche exakt die Funktionalität bietet, die ich benötige, und eine bessere API als ich sie jemals hätte schreiben können. Der Umstieg war recht einfach, da der Transcoder von QtAV lediglich ein Ein- und ein Ausgabegerät benötigt, in diesem Falle eine Datei und eine Netzwerkverbindung, und konfiguriert werden muss. Bei der Konfiguration bin ich auf einige Probleme gestoßen, da das Programm abstürzt oder sich aufhängt, wenn man die Konfiguration nicht auf die richtige Art und in der exakt richtigen Reihenfolge macht. Da QtAV aber einige Beispiele auf GitHub zur Verfügung stellt, konnte ich das Problem in relativ kurzer Zeit lösen. Aktuell bestehen noch Probleme mit der Audio-Übertragung, aber ich denke, dass auch die mit etwas Zeit lösbar sind.

5.1.1 Qt

Qt ist ein Anwendungsframework für C++ und Python, welches nahezu alle Funktionen für die Anwendungsentwicklung bereitstellt, z.B. Datenbank-Anbindung, Netzwerkkommunikation und GUIs. Darüber hinaus gibt es noch diverse Plugins von The Qt Company, also der Firma, die Qt entwickelt und vertreibt, z.B. für 3D-Darstellung, Diagramme etc., sodass man in fast allen Fällen die benötigten Funktionen direkt in Qt findet. Sollte das wie bei mir nicht der Fall sein, existieren zahlreiche Plugins der Community, die häufig Open-Source und gut dokumentiert sind, die man sich dann herunterladen und installieren kann.

In Qt gibt es zwei Möglichkeiten der prozessinternen Kommunikation, zum einen mit Signals und Slots und zum anderen mit Events. Diese beiden sollen im Folgenden kurz erläutert werden, sie in Gänze zu erklären, würde den Rahmen dieser Dokumentation bei weitem sprengen. Wer sich weiter mit Qt beschäftigen möchte, dem empfehle ich die Seite “Qt for Beginners” im Qt Wiki [QfB].

Signal-Slot-Prinzip In Qt gibt es die Möglichkeit, in einer Klasse spezielle Methoden zu deklarieren, die Signale und Slots. Signale muss man nur deklarieren, d.h., sie erhalten keine Funktionalität vom Programmierer. Beim Kompilieren des Codes geht ein Programm namens Meta-Object-Compiler (MOC) durch alle Klassen, die Signale und/oder Slots enthalten oder benutzen und stattdessen diese entsprechend der spezifizierten Verbindungen mit Funktionalität aus, wodurch die Signale und Slots erst nutzbar werden. Diese Funktionalität ist ebenfalls sehr nützlich, wenn man mehrere parallel laufende Funktionen hat, die miteinander Daten austauschen müssen.

Außerdem muss im Programm eine Event-Loop gestartet werden, d.h., es wird eine parallel laufende Funktion gestartet, die nichts anderes tut, als alle in die Event-Queue eingereichten Signale abzuarbeiten und die verbundenen Slots aufzurufen.

Event-Prinzip Auch Events werden von der Event-Loop bearbeitet, daher hat sie auch ihren Namen erhalten. Events werden **nur** bei Oberflächen benutzt, und arbeiten auf einer Ebene unter den Signalen und Slots. Sie werden ausgelöst, wenn der Benutzer im Fenster die Maus bewegt, mit ihr klickt, das Fenster vergrößert oder verkleinert, wenn das Fenster minimiert, maximiert oder in Vollbildmodus gesetzt wird. Es gibt viele weitere Eventtypen, siehe die API-Referenz zu Qt-Events [QER]. Da es nur vordefinierte Events gibt, kann man auch nur vordefinierte Event-Handler überschreiben.

Oberflächen mit QML Für Qt existieren zwei Möglichkeiten eine Oberfläche zu erstellen, eine davon ist die Qt Markup Language (QML). Mit QML kann man sehr schnell Oberflächen erstellen, da es vordefinierte Elemente, wie Knöpfe oder Text-Eingaben, gibt, die nur miteinander kombiniert werden müssen. Außerdem stehen drei Themes zur Auswahl: Material, das Standard Design unter Android 5 - 8, Universal, das Windows App Design, und Fusion, das Standard Qt Design. Direkte Funktionalität lässt sich per Javascript in QML einbetten, zudem ist es möglich, bestimmte, aus C++ zur Verfügung gestellte, Methoden aus QML aufzurufen, sodass Anwendungen mit QML meistens aus zwei Teilen bestehen, dem GUI und der eigentlichen Funktionalität der Anwendung, da C++ deutlich effizienter ist als Javascript. Zwischen QML und C++, wie auch zwischen einzelnen Elementen in QML, erfolgt die Kommunikation meist per Signal-Slot-Prinzip. Die Events werden bei QML-Oberflächen zwar auch generiert, aber von der QML-Engine ausgewertet, sodass in QML keine Events ankommen.

5.1.2 QtAV

QtAV ist ein Plugin für Qt, das Funktionen wie A/V-Wiedergabe/-Transkodierung hinzufügt. Es gibt zwar ein Qt Multimedia Modul von The Qt Company, welches auf den GStreamer Bibliotheken basiert, aber nur die A/V-Wiedergabe und nicht Transkodieren oder Hardwarebeschleunigung unterstützt. Außerdem hat das QtAV-Plugin viel mehr Optionen als das Multimedia Modul, z.B. fehlt im Multimedia Modul die Möglichkeit, festzulegen, auf welche Art man das Video anzeigen möchte, ob in einem separaten Fenster oder so, dass es sich in die bestehende Oberfläche integriert.

5.2 Entwicklungsverlauf

Phase 1	Phase 2	Phase 3	Phase 4
Anforderungen festlegen <ul style="list-style-type: none"> - Festlegen, was Nutzer mit dem Programm machen können - Meilensteine setzen (z.B. Oberfläche funktioniert, Geräte erscheinen, Übertragung an sich funktioniert etc.) 	Entwicklung <ul style="list-style-type: none"> - Meilensteine nacheinander abarbeiten - Veränderte Anforderungen resultieren in anderen Meilensteinen 	Optimierungen und Tests <ul style="list-style-type: none"> - Netzwerkübertragung und Videoverarbeitung auf Effizienz optimieren - Ausgiebiges Testen aller Features, um Bugs etc. zu finden und zu fixen 	Veröffentlichung <ul style="list-style-type: none"> - Lizenz festlegen (GPL, MIT, BSD etc.) - Auf GitHub o.ä. veröffentlichen

5.2.1 Phase 1: Anforderungen festlegen

Zunächst müssen die Anforderungen an die Anwendung festgelegt werden, d.h. es wird festgelegt, was der Benutzer mit der Anwendung machen können soll. Dazu legt man sogenannte User Stories an, die beschreiben, was der Nutzer tun möchte.

Die User Stories entsprechen hier den in der Abbildung erwähnten Meilensteinen.

User Stories

1. Geräte im Netzwerk auflisten lassen
2. Ein Gerät auswählen
3. Datei als Quelle auswählen
4. Ein Profil auswählen
5. Stream starten
6. Stream beenden
7. Stream kontrollieren (Play/Pause, Vor-/Zurückspulen)
8. Profile auf Kompatibilität testen lassen
9. Desktop als Quelle auswählen

5.2.2 Phase 2: Entwicklung

Bei der Entwicklung arbeite ich die Meilensteine der Reihenfolge ab, da Meilensteine immer auf den vorhergehenden basieren. Wenn sich die Anforderungen ändern, z.B. die Möglichkeit den Desktop zu übertragen, werden diese abhängig von deren Voraussetzungen in die Liste Meilensteine eingereiht. Sollte die Anforderung neu sein, dann wird sie an die Liste der Meilensteine angehängt, sollte sich jedoch nur ein bestehender Meilenstein geändert haben, wird dieser zunächst bearbeitet, da es sein kann, dass dadurch die Implementationen der nachfolgenden Meilensteine geändert werden müssen.

5.2.3 Phase 3: Optimierungen und Tests

In dieser Phase ist geplant, die aktuell leistungsaufwändige Transkodierung effizienter zu machen und andere ineffiziente Anwendungsteile durch Tests ausfindig zu machen und zu beheben. Außerdem werden sowohl die gesamte Anwendung als auch die einzelnen Komponenten separat getestet, um Bugs zu finden und zu beheben. Das heißt zum einen, dass ich die gesamte Anwendung, in allen Varianten und Fehlermöglichkeiten testen werden, z.B. dass man während des Streams ein Seite per Taskmanager abbricht. Außerdem werden die einzelnen Komponenten durch Unit-Tests geprüft.

Ein Unit-Test ist eine Sammlung von Methoden, die alle eine Komponente der Anwendung, aber verschiedene Szenarien, prüfen. Man muss so lange Fehler suchen und beheben, bis alle Tests erfolgreich durchlaufen werden.

5.2.4 Phase 4: Veröffentlichung

Zunächst muss ich mich über die verschiedenen Open-Source Lizenzen informieren und eine auswählen. Danach werde ich das Projekt vermutlich auf GitHub veröffentlichen, sodass jeder Zugriff darauf hat und es benutzen kann. Da GitHub auch Features wie Issues bietet, können anderen Fehler oder auch Verbesserungsvorschläge dort einreichen, damit ich diese bearbeiten und gegebenenfalls Rückfragen stellen kann. Außerdem können dann Schulen VanadiumCast nutzen, sobald die Desktop-Übertragung implementiert ist, um den Schülerinnen, Schülern und Lehrkräften eine komfortable Möglichkeit zu bieten, im Unterricht den eigenen Bildschirm zu teilen.

6 Architektur

6.1 Komponenten

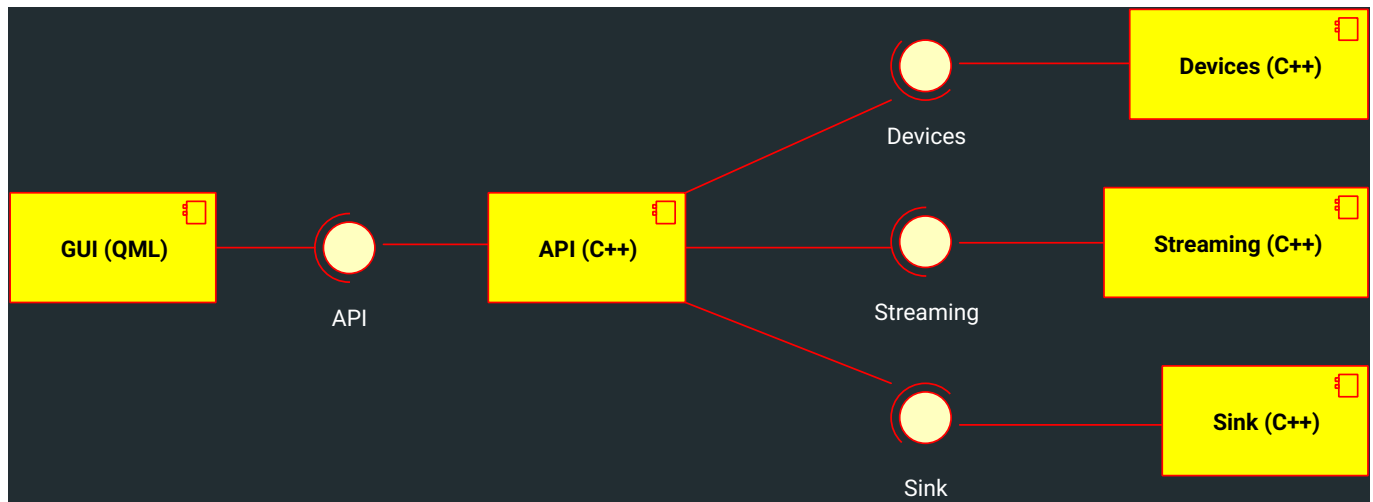


Abb.6.1.1: Komponentendiagramm

Die Anwendung ist so strukturiert, dass sie leicht erweiterbar ist, z.B. kann man die Device-Komponente austauschen, wenn die Geräte per Bluetooth gesucht werden sollen. In der Streaming-Komponente muss in diesem Fall jedoch nichts geändert werden, wenn die Daten weiterhin über das Netzwerk übertragen werden sollen, solange die Geräte ihre IP-Adresse beim Scannen mit zurückgeben. Auch kann man die Sink-Komponente austauschen, wenn man z.B. die Ausgabe in eine Oberfläche integrieren möchte oder der Stream in eine Datei geschrieben statt angezeigt werden soll.

6.1.1 GUI (QML)

Aufbau Die Oberfläche der Anwendung ist in QML geschrieben, daher ist sie vom Backend (Backend $\hat{=}$ alles bis auf die Oberfläche und deren direkt Funktionalität wie "Knopfdruck→Dateiauswahldialog") getrennt und kann auch nur über ein API-Objekt auf die Funktionen des Backends zugreifen. Sie ist nach dem KISS-Prinzip gestaltet (Keep It Simple, Stupid). Der Quellcode der Oberfläche ist in `src/res/gui` zu finden.

Geräteauswahl Hier ist die erste der drei Views zu sehen, die Geräteauswahl. Diese ist, wie jede andere View, so einfach wie möglich gehalten, es gibt nur die Liste der gefundenen Geräte und die nächsten Schritte am unteren Rand. Der nächste Schritt ist jedoch nur auswählbar, wenn ein Gerät ausgewählt wurde, damit der Stream nicht gestartet werden kann, wenn kein Gerät ausgewählt wurde, da dies würde zu einem Absturz des Programmes führen.

Quellenauswahl Zu sehen ist die Quellenauswahl in der aktuellen Fassung mit der Dateiauswahl. Es sind zum einen das Textfeld mit der direkten Eingabemöglichkeit einer URL (lokale Dateien starten mit `file://`) Zudem gibt es einen Knopf zum Öffnen eines komfortableren Dateiauswahldialogs. Nachdem eine Datei ausgewählt wurde, wird sowohl die Videovorschau gestartet als auch der Knopf zum Starten des Streams freigeschaltet, um Fehler zu vermeiden.

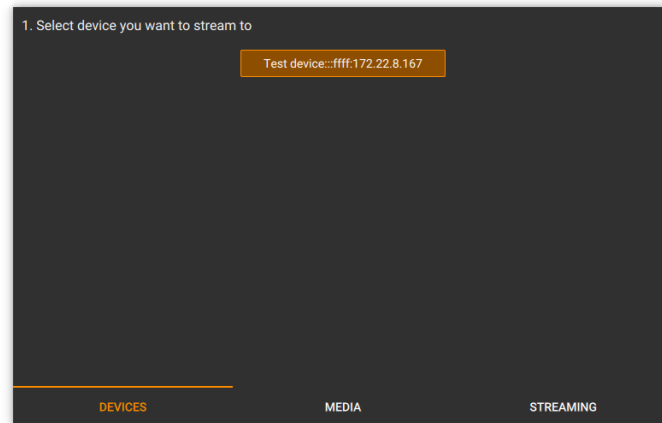


Abb.6.1.1.1: Geräteauswahl

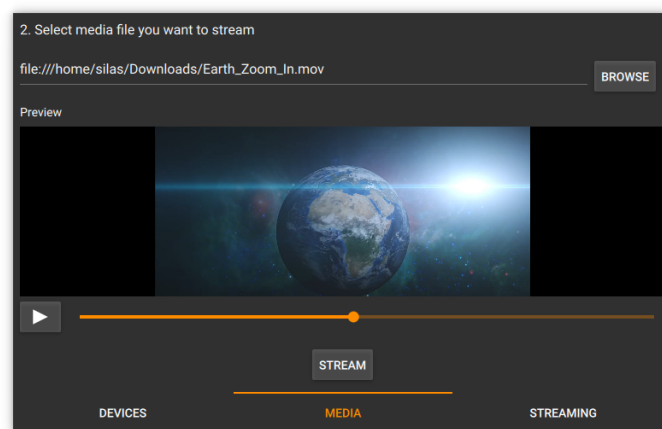


Abb.6.1.1.2: Quellenauswahl

Wiedergabesteuerung Die hier zu sehende Wiedergabesteuerung ist noch konzeptionell und ohne Funktion, d.h. es ist noch möglich, dass sich diese View im Rahmen der Entwicklung noch ändert. Aber auch hier ist das KISS-Prinzip deutlich erkennbar. Es gibt einen Knopf zum Pausieren/Fortsetzen der Wiedergabe, eine Anzeige der aktuellen Position und einen Schieberegler zum Setzen der Position.

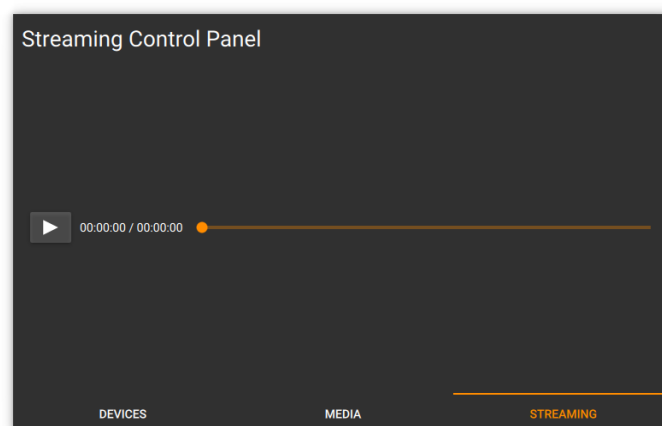


Abb.6.1.1.3: Wiedergabesteuerung (konzeptionell)

6.1.2 API (C++)

Die API die Schnittstelle zwischen QML und dem Backend. Dadurch kann man im Backend sehr einfach Komponenten austauschen, ohne in der Oberfläche etwas ändern zu müssen. In der API ist nur gerade so viel Anwendungslogik vorhanden, dass die einzelnen Komponenten verknüpft werden, der größte Teil ist in den einzelnen Komponenten. In der API ist z.B. die Funktion `startSource()` definiert, die lediglich der zuständigen Komponente Dateipfad und IP-Adresse des Ziels übergibt und die Komponente startet.

6.1.3 Devices (C++)

DeviceScanner Die Klasse `NetworkDeviceScanner` ist für das Scannen nach Geräten im Netzwerk zuständig. Die Funktion `run()` wird parallel zum Rest des Programms ausgeführt und verwaltet die gefundenen Geräte in einer Instanz der Klasse `NetworkDeviceDirectory`. Der Ablauf eines Scan-Vorgangs läuft wie folgt ab:

1. Senden eines UDP-Pakets an den Netzwerk-Broadcast
2. Warten auf eingehende Pakete
3. In jedem eingehenden Paket stehen der Name des antwortenden Geräts und die Senderadresse des Pakets ist die IP des Geräts
4. Jedes so gefundene Gerät wird mit der Funktion `addDevice()` der Klasse `NetworkDeviceDirectory` zum Geräteverzeichnis hinzugefügt (siehe **DeviceDirectory**)
5. Am Ende jedes Scan-Vorgangs wird die Funktion `syncLists()` der Klasse `NetworkDeviceDirectory` verwendet, um die Geräteliste in der Oberfläche zu aktualisieren

DeviceDirectory Die Klasse `NetworkDeviceDirectory` verwaltet Instanzen der Klasse `NetworkDevice`, die in zwei Listen organisiert sind, in der einen stehen die Geräte des letzten vollständigen Scans, in der anderen die Geräte des aktuellen Scans. Diese Trennung ist dazu da, dass in `syncLists()` festgestellt werden kann, welche Geräte neu hinzugekommen sind und welche im aktuellen Scan fehlen. Außerdem wird in `syncLists()` für jedes Gerät, das nur in einer der beiden Listen vorhanden ist, entweder das Signal `addedDevice()` oder das Signal `removedDevice()` aufgerufen, welche mit einem Slot in QML verbunden sind, wo dann die Aktualisierung der Geräteliste in der Oberfläche stattfindet. Durch diese Methode werden nur so viele Aktualisierungen der Oberfläche durchgeführt, wie nötig sind.

Device Ein Gerät wird durch eine Instanz der Klasse `NetworkDevice` repräsentiert. Sie enthält den Namen und die IP-Adresse des Geräts.

6.1.4 Streaming (C++)

VideoTranscoder Die Klasse `VideoTranscoder` enthält den für die Transkodierung zuständigen Code, der im wesentlichen aus dem Initialisieren und Starten des `AVTranscoder` aus dem QtAV-Plugin besteht. Beim Instanzieren der Klasse wird das Streaming-Profil übergeben, welches die Bitrate, FPS, Auflösung und den Videocodec angibt. (siehe Tabelle 6.1.4.1) Aktuell ist das ausgewählte Profil als STANDARD im Code festgeschrieben, es ist aber geplant eine Auswahlmöglichkeit in die Oberfläche einzubauen.

Auflösung	1280x720	1920x1080	1920x1080	2560x1440
FPS	30	30	60	60
Bitrate	1 MBit/s	5 MBit/s	10 MBit/s	15 MBit/s
Videocodec	H.264	H.264	H.264	H.265

Tabelle 6.1.4.1: Streaming-Profile

StreamThread Die Klasse `StreamThread` ist dafür zuständig, Transkoder und Netzwerkverbindung zu initialisieren und anschließend die Daten vom `VideoTranscoder` auf die Netzwerkverbindung zu schreiben. Dazu werden alle 2 ms alle Daten vom `VideoTranscoder` gelesen und auf die Netzwerkverbindung geschrieben. Außerdem steuert diese die Wiedergabe und sendet die entsprechenden Befehle auch an das Ziel, da z.B. erst die Wiedergabe auf dem Ziel pausiert werden muss, weil es sonst zu Fehlern kommt. Zudem wird die Klasse der Oberfläche die aktuelle Wiedergabeposition melden und diese auch von der Oberfläche steuern lassen.

6.1.5 Sink (C++)

SinkHandler Alles, bis auf die Darstellung an sich, ist in der Klasse **NetworkSinkHandler** implementiert, sowohl das Beantworten der Scan-Anfragen mit dem Namen, als auch das Steuern der Wiedergabe und das Senden/Empfangen von Befehlen. Beim Eingang einer Verbindungsanfrage wird zunächst die Oberfläche angewiesen, einen Dialog anzuzeigen, um den Benutzer zu fragen, ob die Verbindung angenommen oder abgelehnt werden soll. Währenddessen wartet der **NetworkSinkHandler** auf die Antwort, wenn diese negativ ausfällt, wird die Verbindung sauber geschlossen, wenn diese positiv ausfällt, dann werden Kontroll- und Datenverbindung aufgebaut, das **readyRead()**-Signal der Kontrollverbindung mit dem Slot **handleControl()** zum Bearbeiten von Kontrollbefehlen wie Pause, Weiter oder Stopp verbunden, der **VideoGuiLauncher** instanziiert und gestartet. Nach dem Absetzen des zeitverzögerten Startbefehls an den AVPlayer, tritt der **NetworkSinkHandler** in eine Schleife ein, die erst beim Beenden des Streams wieder verlassen wird, in welcher auf neue Daten auf der Netzwerkverbindung gewartet wird und diese Daten dann zum AVPlayer weitergeleitet werden.

NetworkSinkTcpServer Der **NetworkSinkTcpServer** ist eine von **QTcpServer** abgeleitete Klasse, die benötigt wird, da die Netzwerkverbindungen auf der Zielseite anders entgegengenommen werden müssen als **QTcpServer** ermöglicht. Dies liegt daran, dass nur der Thread¹, in dem die Netzwerkverbindung erstellt wurde, diese auch nutzen kann. Da sich aber der **QTcpServer** für die Kontrollverbindungen nicht im selben Thread befindet wie der, in dem die **run()**-Methode des **NetworkSinkHandler** läuft, darf die Verbindung erst in der **run()**-Methode erstellt werden, auch wenn sie vorher entgegengenommen wurde.

VideoGuiLauncher Diese Klasse erstellt beim Empfang des Events vom Typ **QEvent::User**, welches für eigene Zwecke benutzt werden kann, das Ausgabefenster und den AVPlayer, welcher gleich darauf gestartet wird. Dieser Vorgang muss durch ein Event ausgelöst werden, da Oberflächenelemente nur im GUI-Thread¹ erstellt, verändert und gelöscht werden können und Events nur im GUI-Thread an Event-Handler² ausgeliefert werden.

¹Ein Thread ist vereinfacht eine parallel zur aufrufenden Funktion laufende Funktion

²Methode, die ein oder mehrere Events bearbeitet