Concepts and Models for Creating Distributed Multimedia Applications and Content in a Multiscreen Environment

vorgelegt von
Dipl.-Ing.
Louay Bassbouss
ORCID: 0000-0001-6801-0924

an der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr.-Ing. Thomas Sikora
Gutachter: Prof. Dr. Manfred Hauswirth
Gutachter: Prof. Dr. Jean-Claude Dufourd
Gutachter: Prof. Dr. habil. Odej Kao

Tag der wissenschaftlichen Aussprache: 12. März 2020

Berlin 2020

Abstract

The continuing trend towards consuming media content on multiple screens such as smart TVs and smartphones is growing steadily. The key enabler for the adoption of multiscreen is the consumption of multimedia content on almost any device with a screen. It is becoming even more significant with the introduction of new media formats such as 360° videos featuring new device categories like head-mounted displays. While traditional application models focus on individual screens, investigations into concepts and models for the provisioning of multiscreen applications and multimedia content across different devices and platforms are only partially addressed. The lack of methods for modeling and conceptualizing multiscreen applications, the requirement for interoperable APIs and protocols, and the need for techniques to deliver high-quality multimedia content to devices with restricted resources are currently the main limitations for a unique multiscreen experience.

This dissertation tackles these limitations and introduces a unified multiscreen application model and runtime environment targeting devices with varying characteristics and capabilities. The proposed approach applies the *Separation of Concerns* design principle to the multiscreen domain. It enables the composition of modular, reusable, atomic, and self-adapting components that can be dynamically migrated between devices within a multiscreen application. Thereby, different communication and distribution paradigms of application components are examined and evaluated.

This work focuses primarily on multimedia applications and presents new techniques for the preparation, distribution, and playback of multimedia content in a multiscreen environment. It proposes a novel approach that enables the playback of processing-intensive content on constrained devices such as the playback of 360° videos on TV sets. The foundation of this approach is the partial pre-rendering of multimedia content and the distribution of the processing load across devices on which the application is running. This also results in a reduction of the required bitrate by up to 80% with the same image quality.

We investigated open web standards as the foundation for the introduced solutions, as the web has quickly developed towards a platform for multimedia applications characterized by rich graphical interfaces and a high level of interactivity across multiple devices and platforms. Some results of this work have been published at international conferences and contributed to the W3C Second Screen Working Group, which defines specifications for multiscreen-related APIs and protocols. Parts of the work have also been patented.

Kurzfassung

Der anhaltende Trend, Medieninhalte auf mehreren Bildschirmen wie Smart TVs und Smartphones zu konsumieren, nimmt stetig zu. Der Hauptfaktor für den Einsatz von Multiscreen ist der Konsum von Multimedia-Inhalten auf fast jedem Gerät mit einem Screen. Sie gewinnt mit der Einführung neuer Medienformate wie 360° Videos und neuen Gerätekategorien wie Head Mounted Displays noch mehr an Bedeutung. Während traditionelle Applikationsmodelle sich auf einzelne Screens beschränken, wurden Konzepte und Modelle zur Bereitstellung von Multiscreen Anwendungen und multimedialen Inhalten über verschiedene Geräte und Plattformen hinweg nur teilweise untersucht. Das Fehlen von Methoden zur Modellierung und Konzeption von Multiscreen Anwendungen, die Anforderungen an interoperable APIs und Protokolle, sowie die Notwendigkeit, hochwertige Multimedia-Inhalte für Geräte mit begrenzten Ressourcen bereitzustellen, sind derzeit die wichtigsten Einschränkungen für ein durchgängiges Multiscreen-Erlebnis.

Diese Dissertation befasst sich mit diesen Einschränkungen und stellt ein einheitliches Multiscreen-Anwendungsmodell und eine Laufzeitumgebung für Geräte mit unterschiedlichen Eigenschaften und Fähigkeiten vor. Der vorgeschlagene Ansatz wendet das *Separation of Concerns* Design-Prinzip auf die Multiscreen-Domäne an. Es ermöglicht die Verwendung modularer, wiederverwendbarer, atomarer und sich selbstanpassender Komponenten, die innerhalb einer Multiscreen-Anwendung dynamisch zwischen Geräten migriert werden können. Dabei werden verschiedene Kommunikations- und Distributionsparadigmen von Anwendungskomponenten erforscht und bewertet.

Diese Arbeit konzentriert sich in erster Linie auf Multimedia Anwendungen und stellt neue Techniken zur Aufbereitung, Verteilung und Wiedergabe von Multimedia-Inhalten in einer Multiscreen-Umgebung vor. Sie schlägt einen innovativen Ansatz vor, der die Wiedergabe von rechenintensiven Inhalten auf Geräten mit eingeschränkten Ressourcen wie der Wiedergabe von 360° Videos auf TV Geräten ermöglicht. Grundlage dieses Ansatzes ist die partielle Prerendering von Multimedia-Inhalten und die Verteilung der Verarbeitungslast auf die Geräte, auf denen die Anwendung läuft. Dies führt auch zu einer Reduzierung der erforderlichen Bitrate um bis zu 80% bei gleicher Bildqualität.

Dazu werden offene Webstandards als Grundlage für die vorgestellten Lösungsansätze untersucht, da sich das Web schnell zu einer Plattform für Multimedia Anwendungen entwickelt hat, die sich durch vielfältige grafische Oberflächen und ein hohes Maß an Interaktivität über mehrere Plattformen hinweg auszeichnet. Einige Ergebnisse dieser Arbeit wurden auf internationalen Konferenzen veröffentlicht und in die W3C Second Screen Working Group eingebracht, die Spezifikationen für Multiscreen-bezogene APIs und Protokolle definiert. Teile der Arbeit wurden patentiert.

Acknowledgement

First of all, I would like to thank my supervisor Prof. Dr. Manfred Hauswirth, who gave me the opportunity to work on this doctoral thesis. I am grateful for his continual openness, patience, and guidance. I would also like to thank Prof. Dr. Jean-Claude Dufourd and Prof. Dr. habil. Odej Kao for their kind support and for reviewing this thesis.

I am also deeply thankful to all my colleagues at the business unit FAME for their support and fruitful discussions over the last years. A special thank goes out to Dr. Stephan Steglich for his continuous support and for giving me the opportunity to do fundamental and applied research in the domain of multiscreen applications and media. I would also like to thank all the students who have graduated under my supervision for their outstanding work.

Finally, I would not have been able to complete this work without the continuous support, encouragement, and warmth of my family. I am grateful for the tremendous support they gave me in the past years to complete this thesis.

Contents

1 Introduction

1.1 Motivation

The majority of our media consumption today occurs in front of a PC, TV, smartphone or tablet. *According to the Google Research Study on Multiscreen User Behavior, "the majority (90 percent) of our daily media interactions are screen based. Only 10 percent of all media interactions are non-screen based (radio, newspaper, and magazine). Smartphones are the most frequent companion devices during simultaneous usage especially when we are watching TV"* [1].

The interaction and collaboration between devices of different types and varying characteristics are becoming increasingly important. For example, Netflix lists more than 25 supported devices in 7 different categories such as Smart TVs and Game Consoles [2] which are also potential candidates to interact with each other especially between mobile and TV devices. The development of multiscreen applications for this heterogeneous landscape of devices and platforms is extremely time and resource intensive. Operators, content providers, and device manufacturers are increasingly looking for solutions that simplify the technologically complex multiscreen landscape and reach consumers on all devices they use in their daily life, regardless of the underlying technologies.

The development of multiscreen applications is facing new challenges that go beyond traditional single-screen applications and requires developers to consider additional aspects such as the discovery of devices, launching applications on remote devices, synchronizing application data and multimedia content across devices, communication between application components, security, and privacy. Therefore, new application development paradigms, models, concepts and technologies that address these challenges are required and worth further investigation.

The main driver for the further development of multiscreen applications is the consumption of Internet-delivered video on all screens. According to the *Cisco Visual Networking Index: Forecast and Methodology, 2016–2021, "Internet video streaming and downloads are beginning to take a larger share of bandwidth and will grow to more than 81 percent of all consumer Internet traffic by 2021"* [3]. Delivering a seamless video experience across different types of devices and platforms is one of the major

challenges for content creators, application developers, distributors and platform providers. Due to the heterogeneous landscape of platforms and extremely varying media rendering capabilities (different formats, codecs, media profiles) even on devices from the same manufacturer, it is almost impossible to distribute the same media content to all user devices in a single format.

Furthermore, the introduction of new media types in recent years, such as 360° video on YouTube [4] and Facebook [5], which is becoming increasingly popular and commercially relevant, adds a new level of complexity to the already complex media delivery landscape. Currently, most 360° videos offer Field of View (FOV) with Standard Definition (SD) resolution, which significantly limits the immersive experience for the user. Bandwidth limitations, end device constraints and lack of higher resolution 360° cameras prevent FOV with better quality to be delivered. Many devices like TVs are also unable to perform the geometric transformation to render the field of view from the spherical video.

Today's multimedia services are either implemented as native software applications, running on a particular device and Operating System (OS) or as web applications served from the Internet (online) and running on a variety of devices and platforms that provide a web runtime. There are already existing open and proprietary solutions that address individual multiscreen features, but unified concepts and models for developing, distributing and running multiscreen applications across multiple devices and platforms are still missing. For example, iOS uses a proprietary protocol called *Airplay* [6] to share content on *Apple TV* [7] via video streaming while Android uses a protocol called *Miracast* [8] for the same purpose. Google Cast [9] follows another concept that allows displaying hosted web content served from the internet on receiver devices like *Chromecast* [10]. In this case, multiple interlinked applications run on host and presentation devices and collaborate with each other. There is also ongoing research on another mechanism that allows to run the application in the cloud and to stream the user interface (UI) or part of it to target devices such as low-cost set-top-boxes (STBs).

1.2 Problem Statement And Research Questions

As outlined in the motivation, the development of interactive multiscreen applications and the creation and delivery of multimedia content across different devices and platforms are highly challenging tasks, and many issues are still not solved or only addressed partially. The problems this dissertation focuses on are:

- **Problem 1**: Lack of a unified method for modeling and conceptual design of multiscreen applications.
- **Problem 2**: Lack of interoperable APIs and protocols for a cross-platform, multiscreen runtime environment.
- **Problem 3**: Lack of techniques for streaming and playback of high-quality multimedia content especially 360° videos on low capability devices with limited resources.

Based on these problems, the following research questions were identified:

- **Research Question 1**: How to design and develop multiscreen applications, taking into account aspects such as development costs and time, platform coverage and interoperability between devices and technology silos?
- **Research Question 2**: How to efficiently distribute and run multiscreen applications, taking into account available resources such as bandwidth, processing, storage and battery without affecting the user experience?
- **Research Question 3**: How to efficiently prepare, stream and play multimedia content, especially 360° videos, across different platforms taking into account available bandwidth, content quality, media rendering capabilities and available resources on target devices?
- **Research Question 4**: How to support the standardization of an interoperable and flexible model for distributed multiscreen applications and the specification of related standard APIs and network protocols?

1.3 Contributions

This work has a strong software engineering focus driven by the substantial contributions to international standards. It aims to solve the problems of "Wild West" architectures and technologies in the open Internet by providing the first structured analysis and classification of methods and concepts for the distribution of applications and media across heterogeneous devices. Based on this structured and methodical analysis, a comprehensive architecture, protocols, and APIs are defined and implemented, and their efficiency is proved by an extensive evaluation process. More specifically, the research contributions of this thesis are:

- **Contribution 1**: Study and evaluate design patterns, concepts, and technologies for creating distributed multimedia applications and identify the requirements to define a unified application model by considering the most relevant multiscreen use cases and application scenarios. The new model defines a set of communicating application components where each of them can be dy-

namically migrated to other devices at any time, and automatically adapted to the target execution context. Thereby, the *Separation of Concerns (SoC)* design principle is applied to this domain by enabling the composition of multiscreen applications from modular and reusable atomic software components.

- **Contribution 2**: Design and development of a multiscreen application framework that reduces the complexity of building and distributing applications across multiple screens and platforms. The core of the framework is a unified, cross-domain and web-based application model that abstracts from the underlying technologies and offers a set of APIs that provide access to core multiscreen functions like device discovery, application launch, synchronization, signaling, and communication. This new framework supports the application model discussed in the first contribution.
- **Contribution 3**: A mechanism to prepare, deliver, and playback multimedia content especially 360° videos even on low-capability devices with limited resources. All methods for processing and streaming of 360° videos such as local rendering, cloud rendering, and the new pre-rendering approach introduced in this thesis are evaluated and compared.
- **Contribution 4**: The results of this thesis are contributed to multiscreen standardization activities in the World Wide Web Consortium (W3C) [11] especially to the *Second Screen Working Group* [12] which works on the two specifications, *Presentation API* [13] and *Remote Playback API* [14], and to the *Second Screen Community Group* [15] which incubates and develops specifications of a network protocol called *Open Screen Protocol* [16].

These contributions were successfully presented at peer-reviewed international conferences by the author of this dissertation:

- **Paper 1:** Louay Bassbouss, Max Tritschler, Stephan Steglich, Kiyoshi Tanaka, and Yasuhiko Miyazaki. „Towards a Multi-screen Application Model for the Web“. In: *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*. Kyoto, Japan, 2013, pp. 528–533. This paper provides the foundation for the development of web-based multiscreen applications. It is expanded in this thesis to support the application model and concepts presented in Chapter 4, using new techniques such as Web components that have been widely adopted by the Web community in recent years. Section 4.5 presents the new enhancements in detail.
- **Paper 2:** Louay Bassbouss, Görkem Güçlü, and Stephan Steglich. „Towards a wake-up and synchronization mechanism for Multiscreen applications using iBeacon“. In: *2014 International Conference on Signal Processing and Multimedia Applications (SIGMAP)*. Vienna, Austria, 2014, pp. 67–72. This paper provides the basis for discovering devices in a multiscreen environment by considering the spatial aspect. The results of this paper are considered in Sec-

tion 4.6.1, which presents the implementation of the multiscreen application runtime introduced in Section 4.4.

- **Paper 3:** Louay Bassbouss, Stephan Steglich, and Martin Lasak. „Best Paper Award: High Quality 360° Video Rendering and Streaming“. In: *Media and ICT for the Creative Industries*. Porto, Portugal, 2016. This paper provides a comparison between the 360° rendering and streaming methods and serves as input for the classification of the different 360° approaches presented in Section 5.3.2.
- **Paper 4:** Louay Bassbouss, Stephan Steglich, and Sascha Braun. „Towards a high efficient 360° video processing and streaming solution in a multiscreen environment“. In: *2017 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*. 2017, pp. 417–422. This paper introduces the 360° pre-rendering approach presented in Section 5.3.4 of this thesis. The new approach enables the playback of 360° videos even on devices with limited processing resources like TVs.
- **Paper 5:** Louay Bassbouss, Stefan Pham, and Stephan Steglich. „Streaming and Playback of 16K 360° Videos on the Web“. In: *2018 IEEE Middle East and North Africa Communications Conference (MENACOMM) (IEEE MENACOMM'18)*. Jounieh, Lebanon, 2018. This paper provides an overview for the creation, delivery, and playback of high-resolution 360° content using the *Dynamic Adaptive Streaming over HTTP (DASH)* standard, and following the pre-rendering approach introduced in the previous paper. The results are presented in Section 5.3.

1.4 Structure of the Thesis

This thesis is structured as follows:

Chapter 2 - State of the Art and Related Work: This chapter includes an overview of related scientific research in the field of this thesis. Also related state-of-the-art technologies and standards in the domain of distributed multimedia applications and content in a multiscreen environment are discussed and evaluated in this chapter.

Chapter 3 - Use Cases and Requirements Analysis: This chapter begins with the definition of use cases that cover most relevant multiscreen application scenarios with focus on mobile and home networked devices like smartphones, tablets, and TVs. Afterward, the defined use cases are analyzed, and the functional and non-functional requirements are identified.

Chapter 4 - Multiscreen Application Model and Concepts: Based on the requirements identified in Chapter 3, this chapter focuses on the definition of a unified model for multiscreen applications and the evaluation of different options for a distributed runtime architecture. Hereby, the different methods for application execution, rendering, and distribution are identified and compared. This detailed analysis is used to derive the technical specifications of the system components and APIs aligned with current standardization work.

Chapter 5 - Multimedia Streaming in a Multiscreen Environment: This chapter focuses on the efficient streaming and playback of high-quality multimedia content with a focus on 360° videos. Based on the use cases and requirements identified in Section 3, the architecture of the playout system is specified, and the different methods for 360° video rendering are studied and evaluated. This chapter provides also a proof-of-concept implementation of most relevant system components enabling the delivery, playback, and synchronization of multimedia content across devices.

Chapter 6 - Evaluation: This chapter evaluates the results of Chapters 4 and 5 according to the functional and non-functional requirements identified in Chapter 3.

Chapter 7 - Conclusions and Outlook: This chapter concludes the thesis with a summary that discusses the achievements of this work against the research questions defined in Section 1.2. Finally, a short outlook on future work and potential follow-up activities are given.

2 State of the Art and Related Work

This chapter discusses state-of-the-art technologies and related work in the fields of distributed multiscreen applications and multimedia streaming. It is structured as follows: Section 2.1 explains the basic terminology and defines the context of this thesis. Afterward, section 2.2 gives an overview of relevant real-world scenarios that motivate the topic of this work. Section 2.3 focuses on state-of-the-art technologies in the field of distributed multiscreen applications and media streaming in general and 360° video streaming in particular. Section 2.4 analyses further scientific activities and research in this field.

2.1 Multiscreen Definition

There are many definitions and interpretations for the term *Multiscreen* depending on the domain being used. There are also alternative or similar terms like *second screen*, *companion screen*, and *dual screen* which are widely used and usually refer to the same concept but for a specific application context. This section will discuss these terms and provide a clear definition of the term multiscreen.

One of the first terms used in this context is *second screen*. In 2007 Cruickshank et al. introduced in a study an approach for enhancing interactive TV services like Electronic Program Guide (EPG) using portable second screen: "*A portable second screen offers the opportunity to remove the need to show UI elements on the main television screen*" [22]. The usage of the second screen at that time was motivated by the limited capabilities of TVs concerning user interface rendering latency and responsiveness. The second screen hype continued in the following years, especially with the launch of the iPad in 2010 accompanied by the increasing number of smartphones and the introduction of HbbTV. Since then, broadcasters have begun to think about valuable real-world scenarios that can enhance the TV experience not only by using the second screen as a replacement for the TV remote control. In Section 2.2, some of the most important scenarios in the broadcast domain will be explained in more detail. At the same time, the trend towards Video on Demand (VOD) has played an important role in the way we consume media. The mobile device began to be the main device attracting our attention and no longer considered as the second screen. The role of TV also began to change from a device for consuming

linear broadcast video to a device featuring personalized OTT content delivered over the Internet. The TV remote control is used in this case to interact with the TV application, mainly to search for new content, control video playback and explore additional information about the current media playing on the TV. It has been proved in [23] that these tasks can be done more easily and quickly with a smartphone or tablet than with a TV remote control. It is also easier to provide a personalized user experience with a personal device like a smartphone than with a shared device like TV. For this reason, big players in the OTT industry especially YouTube and Netflix started to work on methods to connect applications on mobile and TV to provide a better user experience and to elaborate on new application scenarios that go beyond content search and media playback like "multiscreen advertisement".

Another relevant term in the context of multiscreen is *companion screen*, which is widely used in the broadcast domain. It represents devices such as smartphones, tablets, and laptops that can be connected to HbbTV applications provided by the broadcaster.

Dual screen is another term used in conjunction with mobile platforms such as iOS and Android for connecting mobile devices to external displays either wired or wireless to extend or mirror the view of the application running on the host device. These platforms provide SDKs for application developers to display content on the remote display when used in extended mode.

In this thesis, we will consider multiscreen as an umbrella for all these terms and define it as "the participation within a common execution context involving more than one screen with application instances interacting and complementing each other". The application instances can be distributed to devices of different categories and platforms and are not restricted to a specific number of screens or specific device classes. The number of devices involved in a multiscreen scenario can vary during runtime, as screens can be added or removed at any time. Application instances must adapt to the capabilities of the device on which they are running.

Note: multi-screen is another spelling for the term multiscreen and also frequently used in the literature. In this work, we will use multiscreen, but the spelling *multi-screen* may appear when external sources and publications are addressed.

2.2 Motivating Real World Scenarios

There is a variety of multiscreen multimedia applications and services already available. Video streaming is the most relevant category for enhancing the user experience

by using a mobile device while watching a video on the TV. The most popular video streaming services like YouTube [4] and Netflix [24] already support multiscreen. The mobile application allows users to browse the catalog and stream selected content to a connected TV or a streaming device like Chromecast [10] and Apple TV [7]. Providers like Netflix are also experimenting with new features like "Netflix QuietCast" [25], which allows viewers to mute the video on TV and play the audio stream on a companion device while keeping both streams in sync. Social media applications like Facebook [5] also support multiscreen by enabling to cast videos to the big screen while the user can continue to use the app on the mobile device.
Productivity is another important category with relevant multiscreen applications like Google Slides [26] which allows users to display presentation slides on a large screen like Chromecast while using the mobile device as a controller.
Gaming is also one of the important domains for multiscreen applications. The famous mobile game *Angry Birds* [27] was one of the first gaming applications that supported Chromecast. In most multiscreen games, the player uses the smartphone as a game controller while the TV displays the main game field. Some multiscreen games also support multiple players.

Another domain for using multiscreen is *Sport*. During the World Cup 2014 in Brazil, the two German public broadcasters ARD and ZDF extended their second screen offer by showing customizable game statistics in the mobile application while the viewer watched the game on the TV [28]. However, one of the most exciting features of the app was the ability to choose individual camera perspectives from more than 15 cameras distributed in the stadium. *Multiscreen advertisement* is also one of the most important commercially relevant scenarios. Services like wywy, which support hundreds of TV channels in Europe and the US, and create a seamless brand experience through multiscreen advertisement by analyzing the TV audio signal to recognize the content and display complementary and interactive ad content on the TV. A user study with a Nissan TV synced campaign led to 96% brand uplift [29]. Other providers like Shazam [30] also use content recognition techniques with audio watermarking for real-time synchronization between TV and companion screen.

Many broadcasters also see social TV and storytelling applications as a way to engage viewers with the content displayed on TV. The *Walking Dead Story Sync* [31] is one of the most popular storytelling second screen applications in the USA for the TV series *Walking Dead*. This kind of applications also offers social media integration, which allows viewers to engage with friends about the current TV program.

Also, many of the VOD and social media services such as YouTube and Facebook are now supporting 360° videos on various devices like head-mounted displays. Many broadcasters and content providers such as ZDF [32], Arte [33] and RedBull [34] have created their own 360° content and made it available to viewers via mobile

applications. 360° video on TV is still limited and only supported in the YouTube application [35] on a few new Android TV models.

2.3 State of the Art Technologies and Standards

This section discusses state-of-the-art technologies and standards that are relevant for the multiscreen multimedia application domain. Different aspects will be considered in each of the following sections.

2.3.1 Discovery, Launch and Control

In this section, existing protocols and standards for discovering devices or services, launching and controlling applications and media on secondary devices will be discussed.

SSDP

The Simple Service Discovery Protocol SSDP [36] is a network protocol for advertising and discovery of network services without the need for a server-based configuration to register the services and also without the need for a special static configuration of a network host. SSDP is the discovery layer of the UPnP protocol, but it is often used in other technologies and standards like DIAL and HbbTV as a standalone discovery protocol without the other UPnP components. SSDP is a text-based protocol that uses UDP [37] as the underlying transport protocol. It can be implemented on any platform that supports UDP sockets. All SSDP Service announcement and discovery requests are sent to the multicast address `239.255.255.250` (IPv4) and port `1900`.

UPnP

The Universal Plug and Play protocol UPnP [38] defines an architecture for ad-hoc and peer-to-peer connectivity in small and unmanaged networks. The UPnP architecture includes other protocols like TCP, UDP, HTTP, and XML. UPnP also requires that a device has been assigned an IP address. Besides addressing, UPnP architecture includes the following layers:

- **Discovery**: UPnP uses SSDP as a discovery protocol.

- **Description**: After a device or service is discovered, control devices can retrieve its device description from the `LOCATION` URL provided in the discovery response message.
- **Control**: When a control device receives and parses a device description, it can connect and control a specific service offered by that device.
- **Eventing**: In many situations, it is necessary to send update events to control devices. For example, a media rendering device can send events about the current status and playback position to control devices in order to update the control UI.

DIAL

The Discovery and Launch protocol DIAL [39] developed by Netflix [24] allows second screen devices like smartphones and tablets to discover and launch applications on TVs and streaming devices. The DIAL protocol reduces the number of steps needed to connect an application running on a second screen to its counterpart application running on the TV. There is no need for end-users to enter PIN codes manually or scan QR codes to pair the devices together. DIAL specifies client and server components for first and second screen devices. The DIAL server exposes a service in the network that provides interfaces for launching, stopping and checking the status of a specific application. DIAL clients discover devices exposing DIAL services in order to launch or stop TV applications. Application names like YouTube and Netflix are used as identifier. To avoid conflicts, providers need to register the names or namespaces of their applications in a DIAL registry. Similar to UPnP, DIAL also uses SSDP as underlying discovery protocol.

mDNS/DNS-SD

The multicast Domain Name System mDNS [40] and the DNS-based service discovery DNS-SD [41] are two protocols that can be used in conjunction with each other to support network service discovery.

- **mDNS**: The mDNS protocol uses APIs similar to the unicast Domain Name System, but it relies on multicast UDP protocol. It enables the lookup of DNS resource records without the need for a conventional managed DNS server. Each device in the network stores a list of DNS resource records and joins the mDNS multicast group by sending requests and listening to the multicast address `224.0.0.251` and port `5353`. mDNS defines a top-level domain `.local` for local addresses. When a client needs to resolve a hostname, it sends a

request to the multicast address and asks the host with that name to identify itself. The host sends a multicast message with its IP address. All devices in the multicast group also receive the message and update their cache. When a device disappears, it sends a multicast message with Time To Live header `TTL=0`. All devices in the multicast group receive the message and remove that device from the cache.

- **DNS-SD**: It extends mDNS to provide simple service discovery and not only advertising and resolving hostnames. Similar to SSDP, DNS-SD provides functions to advertise and discover services in the network. It allows clients to discover a named list of services from a specific type using the *DNS PTR* record. A service instance can be described using *DNS SRV* and *DNS TXT* records.

Google Cast

Google introduced the Google Cast SDK for the platforms Android, iOS and Web which allows developers to stream content to Cast devices such as Chromecast and Android TV. The Google Cast Protocol used behind the Cast SDK enables sender and receiver devices to discover, control and communicate with each other. The first version of Google Cast used DIAL for discovery and launch of applications and a proprietary socket-based protocol for communication. The latest version of the protocol also supports mDNS/DNS-SD for discovery and a proprietary socket-based protocol for application control. Cast receiver applications incorporate HTML5 technologies and can be hosted on any Web Server and updated any time.

HbbTV

The Hybrid broadcast broadband TV HbbTV [42] is a global initiative aimed at harmonizing the broadcast and broadband delivery of entertainment services to consumers through connected TVs, set-top boxes and multiscreen devices. HbbTV has a wide range of supporters, especially from European broadcasters and consumer electronics manufacturers. The consortium published recently the version 2.0.2 of the HbbTV specification which includes a set of new features like HTML5, CSS3, HEVC, DASH, Companion Screens, and Media Synchronization. The main functions of the Companion Screens and Media Synchronization components are:

- **Launching a companion screen application**: allows an HbbTV application to launch a second screen application on a companion device using the `HbbTVCSManager` interface.

- **Application to application communication**: allows second screen and HbbTV applications to establish a communication channel using WebSocket [43]. The HbbTV terminal runs a WebSocket server and each of the second screen and HbbTV applications connects to that server and joins the same session.
- **Remotely launching an HbbTV application**: allows a second screen application to join or launch an HbbTV application from a companion device using DIAL. The HbbTV terminal runs a DIAL server that offers an application called HbbTV that is responsible for handling all HbbTV related requests. The DIAL server may be used to launch other applications like YouTube which is not in the scope of the HbbTV specification.
- **Multi-Device Synchronization**: allows synchronizing data and media streams delivered over broadcast or broadband between companion devices and HbbTV terminals. It also allows synchronizing audio and video streams on the same terminal.

BLE based discovery

Bluetooth Low Energy BLE [44] (also called Bluetooth Smart) provides a power-friendly solution to discover devices nearby. A service device transmits during its operating time a BLE packet also called Beacon containing information that can be used to identify the device. Control devices listen to beacons from a specific type and notify applications if users enter or leave a region of a beacon. The Bluetooth signal strength can also be used to estimate the distance to the service device. There are two popular technologies on top of BLE introduced in recent years:

- **iBeacon**: is a special format of BLE Beacons introduced by Apple [45] that allow devices or sensors to transmit beacons that contain in addition to the BLE packet headers three main parameters `proximityUUID`, `major` and `minor`. `proximityUUID` is A 128-bit value that uniquely identifies one or more beacons as a certain type or from a certain organization. `major` is a 16-bit unsigned integer that can be used to group related beacons that have the same `proximityUUID`. `minor` is a 16-bit unsigned integer that differentiates beacons with the same `proximityUUID` and `major` value. iOS devices with integrated BLE sensors already support iBeacon and allow to wake up applications and run them in the background for a limited time when the user enters or leaves a beacon region. The application needs to register itself for beacons with specific `proximityUUID` to use this technology.
- **Eddystone**: is also a protocol based on BLE introduced by Google as part of the Physical Web [46] project using a special format of BLE beacons. It is more open than iBeacon since it broadcasts URLs or URIs that can be consumed by

any web browser. This is an advantage compared to iBeacon, where a native application must be installed to receive and interpret the beacon. Eddystone BLE packets contain header parameters and the encoded URL with a length up to 18 bytes. URL shortener services can be used if the original URL cannot be encoded in less than 18 bytes. Eddystone Browsers listen to beacons and retrieve additional information like title, description, and icon of the web page behind the Eddystone URL.

BLE based protocols like iBeacon and Eddystone can be used in a multiscreen application to discover and pair devices based on their proximity.

2.3.2 Screen Sharing and Control

The following sections introduce state-of-the-art technologies and standards for Screen Sharing and Control across devices and platforms.

Airplay

Airplay [6] is a streaming protocol supported on Apple platforms like iOS, macOS, and tvOS. iOS provides an SDK that hides the complexity of the protocol for developers. Airplay can be operated in two modes "Media Sharing" and "Screen Sharing":

- *Media Sharing*: allows to share and control media content like audio, video, and image on Airplay-enabled receivers like AppleTV. This feature is available in safari browser for HTML media elements and can be activated using the `x-webkit-airplay="allow"` attribute.
- *Screen Sharing*: enables screen mirroring or extension on Airplay-enabled receivers. In extension mode, the application can render any content on the connected Airplay device using the Airplay SDK.

The Specification of the Airplay protocol is not public, but there are different Airplay server and client implementations of the protocol which are based on the *Unofficial AirPlay Protocol Specification* [47].

Miracast

Miracast [8] is a peer-to-peer wireless screen sharing standard formed via *Wi-Fi Direct* connections without a wireless access point. It allows client devices like laptops, tablets, and smartphones to stream audio and video content to Miracast-enabled receivers like TVs and projectors. Miracast is effectively a wireless HDMI cable, copying everything from one screen to another using the H.264 codec and its digital rights management (DRM) layer emulating the HDMI system. Miracast is already supported on a wide range of devices like Android smartphones and tablets (version 4.2 and higher), Windows PCs, projectors, TVs, Set-Top-Boxes and game consoles. Older devices can also be extended with Miracast adapters which can be plugged into the HDMI input of any display device.

MHL

The Mobile High-Definition Link MHL [48] is an industry standard which allows sharing screen content of a mobile device like a smartphone or tablet on large screens like high-definition TVs while charging the device. It is an adaptation of HDMI intended for mobile devices. MHL also supports interactions using the Remote Control Protocol RCP. In this case, users can use the TV remote control as an input device instead of the touchscreen which is suitable for video-centric applications. The MHL 3.0 standard supports up to 4K (Ultra HD) video and 7.1 surround-sound audio. MHL 3.0 also supports simultaneous high-speed data channel and improved RCP with new commands.

2.3.3 Application to Application Communication

Application to application (App2App) communication is one of the essential features when developing multiscreen applications. It allows application components distributed on different devices to interact with each other in order to share content or synchronize application state and media streams. The following sections introduce state-of-the-art technologies that are relevant for App2App communication in a multiscreen environment.

IP-based Communication Protocols and their counterpart W3C APIs

IP-based protocols are most often used for the communication between multiscreen application components especially when it comes to Web-based applications that can

make use of these protocols using standard W3C APIs. The most relevant protocols and their counterpart W3C APIs are described below:

- **HTTP**: The Hypertext Transfer Protocol (HTTP) [49] forms the foundation for communication on the Web. It is a Request/Response protocol widely used in distributed systems based on the Client/Server computing paradigm. HTTP is an application protocol and often uses TCP as a transport protocol, but it is not necessarily limited to it. For example, the new state-of-the-art transport protocol QUIC (Quick UDP Internet Connections) [50] can be used instead of TCP. QUIC reduces the latency in the communication and the connection establishment costs compared to that of TCP and supports all new features introduced in version 2 of the HTTP protocol. QUIC is already supported in the Chrome Browser and will be automatically selected instead of TCP if the server that hosts the requested resource also supports QUIC. Since HTTP is a Request/Response protocol, it is not suitable for multiscreen App2App communication and used in most cases as a fallback to other more suitable protocols like WebSocket and WebRTC. A proxy server is needed to enable App2App communication over HTTP. It acts as a relay by sending data back and forward between the application components. Thereby, long polling is used as a mechanism that allows the server to push data to the clients. In Web runtimes like browsers, the XMLHttpRequest API [51] and the new Fetch API [52] can be used to access resources on the server using HTTP.
- **WebSocket**: WebSocket [43] is a bi-directional communication protocol between client and server. The protocol aims to enable the server to push data to the client without establishing a new connection like in the case when using HTTP with long polling. By using WebSocket, the client establishes a single TCP connection to the server which can be used to send data in both directions multiple times. For App2App communication, each application needs to establish a WebSocket connection to the server which forwards the data between paired connections. HbbTV follows this concept for the communication between companion screens and TV terminals by running a WebSocket Server on the TV. The WebSocket API [53] can be used to establish a WebSocket connection from a web page running in a browser or any web runtime.
- **WebRTC**: WebRTC [54] is a peer-to-peer protocol that enables real-time communication (RTC) between web applications via simple APIs. Most relevant scenarios for using WebRTC are messaging, video chat and file transfer applications without the need to install browser plugins and run server infrastructure since the data transfer is done directly between the peers without the need for a proxy. The W3C WebRTC API [55] can be used in browsers and web runtimes to access the underlying WebRTC protocol. The most relevant components of the API are listed below:

MediaStream API: enables the access to local multimedia devices like microphone and camera. The API also allows to capture the screen and use it as a media source.

RTCPeerConnection API: enables the exchange of media streams and data between browsers. Exchanging signaling messages during the connection setup phase is necessary. How to exchange these messages is not part of the WebRTC protocol. Developers can use existing signaling protocols like XMPP or WebSocket for this purpose. Once the connection is established, both peers can add media sources to it or create data channels. Once a media source is added on one end, the media stream can be consumed on the other end using HTML *video* and *audio* elements.

RTCDataChannel API: enables the peer-to-peer exchange of arbitrary data, with low latency and high throughput.

WebRTC is already supported in major browsers like Chrome, Firefox, Opera and Safari on Desktop and mobile platforms. WebRTC is an appropriate App2App communication protocol in a multiscreen environment since no central server is required to transmit the data. There is even no need for a signaling server in local networks since the signaling messages for establishing the WebRTC connection can be exchanged using network discovery protocols like SSDP or mDNS/DNS-SD.

- **Wi-Fi Direct**: Wi-Fi Direct [56] is a standard that enables the direct communication between devices without a wireless access point. Screen Sharing like Miracast is one usage scenario for Wi-Fi Direct. It uses the same Wi-Fi technology for communicating with wireless routers. A Wi-Fi Direct device can essentially function as an access point, and other Wi-Fi-enabled devices can connect directly to it. Wi-Fi Direct also supports device and service discovery. For example, a control device can search for receiver devices that support only Miracast Screen Mirroring. Currently, there is no W3C API which allows web applications to make use of this technology.

2.3.4 Media Delivery and Rendering

Media Delivery and Rendering play an essential role in multiscreen multimedia applications. The challenge is to provide a smooth media playback on devices with varying screen resolutions, network connectivity, and media rendering capabilities. Also, new formats like 360° videos add a new level of complexity since the rendering of 360° content requires additional processing resources and network bandwidth. The following sections discuss state-of-the-art technologies and standards for video codecs, media streaming principles, and 360° video rendering.

Video Codecs

Delivering video content to devices with varying media capabilities requires to identify the best video codec and profile for each supported device and platform. For example, the H.265 and VP9 video codecs are more suitable for UHD 4K content than H.264 video codec since they save between 30%-50% of the bitrate with the same output quality. In many situations, the media content needs to be encoded with different codecs in case there is no common codec supported on the devices under consideration. There are even different profiles for the same video codec which are suitable for a specific resolution, bitrate, and framerate. State-of-the-art video codec technologies are discussed below.

- **H.264**: The H.264 or MPEG-4 Part 10, Advanced Video Coding (MPEG-4 AVC) [57] is an industry standard for video compression first published by ITU-T and ISO/IEC in 2003. H.264 defines a set of profiles which corresponds to a set of capabilities targeting a specific class of applications like OTT, video conferencing and TV broadcast. The standard also defines levels in the same profile which indicate the required decoder performance. H.264 is the video codec with the most coverage across all platforms. A multiscreen multimedia application can provide the video content in H.264 in order to support most devices and platforms. Other codecs can also be provided and dynamically selected if they are supported on the target platform.

- **H.265**: The H.265 or High Efficiency Video Coding (HEVC) [58] brings around 30%-50% better compression with equal video quality comparing to H.264. It also supports resolutions up to 8K UHD. The adoption of H.265 is still low on the Web. Currently, Edge and Safari Browsers support H.265. Therefore, it is recommended to use H.265 together with H.264 in a multiscreen application. If a target device cannot play the H.265 content, it will switch to the H.264 version.

- **VP9**: VP9 [59] is an open and royalty-free video codec developed by Google and is a competitor of HEVC. It has more Browser support than HEVC but still not at the same support level as H.264. Safari is one of the major browsers that does not support VP9. Large-Scale comparison of the three codecs H.264, H.265, and VP9 done by Netflix [60] showed that H.265 and VP9 save 53,3% and 42.6% bitrate compared to H.264 for a video resolution of 1080p.

- **AV1**: AOMedia Video 1 (AV1) [61] is also an open and royalty-free video codec developed by the Alliance for Open Media which was founded by leading

Internet companies like Google, Netflix, and Amazon. AV1 is the successor of the VP9 codec and will replace it in the future.

As can be seen from this overview, the media codec landscape is complicated and fragmented. It is recommended to use H.264 as a common video codec in video-centric multiscreen applications. Other video codecs can be provided on top and can be selected instead of H.264 if they are supported on the target device.

File and Container Formats

In the previous section, we discussed the most important video codecs. In this section, we will focus on container formats which define how video data and metadata coexist in media files. The client program needs to understand the container format and to be able to decode the video and audio data in it. The most relevant state-of-the-art container formats are introduced below:

- **ISOBMFF**: The ISO base media file format (ISOBMFF) [62] developed by ISO/IEC specifies the file structure of metadata and media content. It is one of the most important file formats for media delivery and playback on the web. The W3C Media Source Extension API (MSE) [63] works on top of ISOBMFF which supports a variety of codecs like H.264, H.265, and VP9.

- **MPEG-TS**: The MPEG Transport Stream (MPEG-TS) [64] is another container format also developed by IS0/IEC. It is widely used in Broadcast streaming and supports different codecs like H.264 and H.265.

- **CMAF**: ISOBMFF and MPEG-TS are the two most used container formats for streaming OTT content over the internet. The reason for this is because ISOBMFF is the standard container format for DASH and MPEG-TS is the standard container format for HLS and both are the two dominant Adaptive Bitrate streaming technologies for media delivery over the internet. In order to reach all user devices, content providers need to create the content for both formats which requires additional storage and processing resources. Also, CDNs need to cache two different versions of the video which contain the same video data. The Common Media Application Format (CMAF) [65] was recently introduced as a common container format for DASH and HLS. CMAF will play an important role in video-centric multiscreen applications in the future since the media content will be available in a single format which will reduce the storage and streaming costs.

- **OMAF**: the Omnidirectional Media Application Format (OMAF) [66] is a new container format under development for VR content such as 360° videos. It uses ISOBMFF as file format and provides all the metadata required for interoperable rendering of 360° monoscopic and stereoscopic videos. The metadata may include, for example, the type of the projection used in the 360° video.

Adaptive Bitrate Streaming

Adaptive Bitrate (ABR) streaming is a technique for streaming media content to user devices in an efficient way and the best usable quality under specific conditions. The most relevant factors that are considered in ABR streaming are the available bandwidth, device resolution, and video decoding capabilities. The basic idea of ABR streaming is to spit media content into small video segments (a segment has a duration of few seconds) and make them available in different bitrates, resolutions and maybe in different codecs. The client implements the entire player logic for accessing video segments in the best possible quality and play them back in the correct sequence. The client monitors the network bandwidth and adapts to changes by selecting higher or lower bitrates according to the newly available bandwidth. ABR brings many advantages compared to progressive video streaming where the media content is provided in single files that can be downloaded and played back but without any adaptation to the device and network capabilities. The best known ABR streaming standards are listed below:

- **DASH**: The Dynamic Adaptive Streaming over HTTP (DASH) [67] is a streaming protocol that allows video players to switch between different video bitrates based on different metrics like network performance. It also allows the player to select the appropriate video segments based on device capabilities like display resolution and supported video codecs. Video segments are usually delivered via HTTP. The entire player logic including buffering strategies is implemented in the client. Content Delivery Networks (CDNs) are used to provide high availability of the content. The most important part of DASH is the Media Presentation Description (MPD) manifest which is an XML based document containing all information a client needs to play a video like the location of media segments, supported codecs and available bitrates. Some devices provide native DASH support, but most implementations for web Browsers like *dash.js* are based on the W3C Media Source Extension MSE. DASH works with different container formats and video codecs, but most DASH profiles specify ISOBMFF as a container format.

- **HLS**: HTTP Live Streaming (HLS) [68] is also a streaming protocol from Apple providing the same features as in DASH. HLS uses MPEG-TS as a container format, but it is codec agnostic. Similar to DASH, HLS defines a manifest format called *m3u8* which defines the available bitrate levels and the video segments associated with each level. Many browsers like Safari, Edge and Chrome for Android support HLS natively. It is also possible to play HLS videos in browsers that support MSE by transmuxing MPEG-TS segments into ISOBMFF segments which are supported in the MSE API. *hls.js* is an open source project that provides an HLS player implemented on top of MSE.

360° Video Rendering

The production of 360° videos comprise several steps starting from capturing the video content to stitching, encoding, delivery, decoding and rendering on the client. Usually, a 360° video is captured with multiple wide-angle cameras with overlapping field of views. Their content is put together to produce a single video. This process is called stitching. The output of the stitching is a regular video where the video frames are created from the captured content by applying a specific projection. The most important projection formats are listed below:

- **Equirectangular projection** [69]: It is the most used and most common projection in 360° video production where latitudes and longitudes are used to form a square grid. This type of projection is easy to visualize on a plane, and the output is rectangular which allows the software to encode it in a regular video and stream it using existing delivery infrastructures. On the other hand, this type of projection has some disadvantages. First, the poles of the projection get a lot more pixels than the equator which results in a higher bitrate for the same quality due to redundant pixels. Secondly, 360° videos produced with equirectangular projections have a high distortion which makes the video compression harder compared to regular videos.
- **Cube map projection** [69]: The idea behind the cube map projection is to project portions of the videos behind the six faces of a cube. It is used frequently in the gaming industry to create skyboxes [70]. There are a few benefits of using a cube map instead of the traditional equirectangular projection: *Cube maps don't have geometric distortion and each face looks exactly as if the viewer is looking directly at it with a perspective camera that warps or transforms an object and its surroundings. This is important because video codecs assume motion vectors as straight lines. And that's why it encodes better than with bended motions in equirectangular videos* [71]. Another advantage of the Cube Map

projection is that the output video bitrate can be reduced since there are no redundant pixels as in the equirectangular projection.

- **Pyramid Projection** [72]: The Pyramid projection is about *putting a sphere inside a pyramid so that the base of the pyramid is the full-resolution FOV and the sides gradually decrease in quality until they reach a point directly opposite from the viewport, behind the viewer* [72]. The sides of the pyramid are stretched to fit the entire 360° image into a rectangular frame, which reduces the file size by 80 percent against the original. In order to preserve the quality when the viewer changes the perspective, multiple videos with different viewports need to be generated. *In total, there are 30 viewports covering the sphere, separated by about 30°* [72]. This increases the storage costs comparing to the equirectangular and cube map projections. On the client side, the player jumps between the videos depending on the view angle of the viewer.

After the content is delivered to the client, the 360° video player needs to process each video frame based on the current viewing angle to calculate the FOV image and display it to the user. To perform the geometrical transformation, there are some requirements on the graphical processing capabilities; otherwise, the user experience will suffer. Another implication that may also affect the performance of the client is the high resolution and bitrate of the source 360° video which is mostly produced in 4K resolution and results in a FOV resolution between SD and HD depending on opening angle of the FOV. Some APIs like WebGL and HTML5 Canvas are required in order to render a 360° video in a browser environment. WebGL is available in all modern browsers on desktop and mobile and offers a set of functions implemented on top of OpenGL. Input devices like keyboard, mouse, touch or gyroscope can be used to control the FOV. The motion-to-photon latency on head-mounted displays must be under 20ms to avoid motion sickness. The new WebXR Device API [73] which is still under development addresses these requirements. *This specification describes support for accessing virtual reality (VR) and augmented reality (AR) devices, including sensors and head-mounted displays, on the Web* [73].

2.3.5 Web APIs

In a web runtime, a multimedia application can access underlying system functions using a set of Web APIs. These APIs are standardized or still under development in different W3C standardization groups. The author of this thesis is actively involved in the Second Screen Working Group [12] where the Presentation API [13] and the Remote Playback API [14] are standardized. The author has contributed research results in the field of multiscreen applications, in particular, the results published in [17].

W3C Presentation API

The W3C Presentation API defines a specification that allows web pages to display web content on presentation devices like TVs and to establish a communication channel between the pages running on the different devices. The specification is part of the W3C Second Screen Working Group and focuses only on the application interfaces but abstracts from the underlying protocols for discovery, launch, and communication.

W3C Remote Playback API

The Remote Playback API is another specification of the W3C Second Screen Working Group. Through the W3C Remote Playback API, it is possible with a few lines of code to cast a video or audio from a web page to a presentation device in the same network. Furthermore, it allows the player on the host device to control the media playback on the presentation device. It also offers a mechanism to synchronize the video timeline and playback state between the host and presentation devices.

W3C Web Media APIs

In the first generation of web browsers, the only method to play media content was using third-party plugins. The situation has changed in recent years, and all browser vendors already support media playback using the HTML video and audio elements on all platforms. Most browser vendors have also removed support for third-party plugins. The specification defines the interfaces and events of the HTML Video and Audio elements, but it does not force browser vendors to use a specific codec or container format. Most browsers support H.264 video codec and mp4 container, and some of them also support adaptive bitrate streaming formats like DASH and HLS. Browsers with no native support for adaptive bitrate streaming provide the Media Source Extension API (MSE) [63] which allows implementing DASH and HLS only using JavaScript. *hls.js* and *dash.js* are two widely used open source libraries that implement HLS and DASH. The MSE specification is currently a W3C recommendation and offers functions that allow web applications to append, replace or remove video segments to/from the media buffer.

2.4 Related Work

This section covers other work and research activities related to applications and multimedia content in a multiscreen environment. The following sections explore each of these areas and weigh them against the expected results of this thesis.

2.4.1 Multiscreen Applications

Igarashi et al. propose in the paper "*Expanding the Horizontal of Web*" [74] an approach that allows web applications to interact with home-networked devices like Smart TVs. It proposes a "*Network Device Connection API*" that allows web pages to stream media content to UPnP devices [38]. A similar approach is introduced in the paper "*A Multi-protocol Home Networking Implementation for HTML5*" [75], which focuses on the discovery of home-networked devices from web applications using UPnP and mDNS [40] through a Java Applet that acts as an interface between JavaScript and the networking layer. The "*W3C Network Service Discovery API*" [76] with initial draft specification published in 2012 follows a similar approach. Only one browser vendor has implemented this API in experimental mode, but for security and privacy concerns never utilized it in a production deployment. For example, if a user allows a Web page to access the API, it will be able to find and access any home networked device and create a fingerprint of the user. According to the latest status update from January 2017, the work on the API has been discontinued.

Baba et al. introduced in the paper "*Advanced Hybrid Broadcast and Broadband System for Enhanced Broadcasting Services*" [77] a technology called Hybridcast which aims to integrate broadcast and broadband technologies to provide a better user experience for linear and on-demand TV. Hybridcast enhances existing broadcast services with additional broadband services on TV and mobile. It provides the required components for the linkage of mobile and TV devices, communication, and synchronization of content across devices. Hybridcast applications are web applications with additional JavaScript APIs to access features like launch and communication with second screen applications. The launch process of a companion application requires cooperation between the broadcaster and the TV manufacturer for injecting a manufacturer specific JavaScript library in the broadcaster companion application. This solution has some disadvantages concerning security and makes the development of applications more complex and dependent on the device manufacturer.

Imoto et al. introduced "*A Framework for supporting the development of Multi-Screen Web Applications*" [78]. The framework aims to simplify the development of multiscreen applications by using web technologies like HTML, JavaScript, and

CSS. The main approach of the framework is to allow developers to implement single web applications without dealing with core multiscreen aspects like discovery, communication, and synchronization. The runtime environment consists of a user agent that runs the application within a single execution context in the cloud and distributes parts of the DOM tree to connected devices. The client is a JavaScript library that runs in the Browser and connects to the corresponding application running in the cloud. The client gets a copy of a particular part of the DOM tree that corresponds to the device on which the client is running. The framework keeps copies of the DOM on all devices in sync. All user inputs like keyboard, mouse, and touch are sent to the cloud application and triggered on the corresponding elements. The advantage of this solution is to accelerate the development of multiscreen applications while reducing development costs. On the other hand, the framework has many limitations regarding the support of different capabilities, access to device APIs, offline usage, and synchronization of HTML elements that are not under DOM control like video and canvas elements.

Song et al. introduced in a paper a "Multiscreen Web App Platform" called Pars [79]. "*A Pars web app consists of components that can run distributed on a set of devices as if they are running on a single device*". The Pars platform consists of a daemon which runs on each device and enables the discovery of devices and the communication between them. It also consists of a JavaScript library which allows Pars applications to interface with the underlying network layer. The Pars framework focuses on migrating parts of a web application to other Pars devices in the network while keeping them in sync using a coordination component. It follows a similar approach as proposed by Imoto et al. [78] but without the need for a central entity running in the cloud.

Kim et al. discussed in the paper "*Partial Service/Application Migration and Device Adaptive User Interface across Multiple Screens*" [80] different approaches for full and partial migration of applications across devices. The full migration of an application from one device to another restores the state and functions of the application on the target device and closes the connection to the source device while the partial migration moves or replicates part of the application on the target device without closing the connection. The paper also addresses the need for a mechanism to adapt migrated applications to I/O capabilities like screen resolution and input method. The paper does not provide a solution on how to implement app migration rather than just discussing the use cases that can be enabled by it.

Thomsen et al. introduced in the paper "*Linking Web Content Seamlessly with Broadcast Television: Issues and Lessons Learned*" [81] a platform called LinkedTV that aims to enrich broadcast content with additional information available on the Web and displayed on the second screen in sync with the TV. The paper focuses mainly on

the way how to address and identify specific parts of the broadcast program using the *Media Fragment URI* specification [82]. It uses an open annotation model to augment parts of TV content with annotations. The LinkedTV provides a generic solution for specific use cases which can be adopted across different broadcasters.

Borch et al. introduced "*An architecture for second screen experiences based upon distributed social networks of people, devices and programs*" [83] which describes a new idea accompanied with use cases for encouraging TV viewers to use the second screen to enable interactive social experiences. This solution differs from other solutions in the fact that it uses social media networks as an underlying platform for discovering and connecting devices. A device can share its presence information together with information about the location with other friend's devices. A location can be determined from the public IP address, GPS or via Bluetooth beacons. The paper also describes how the UI of the application can be synchronized across devices based on "*dynamic measurement of network latency and delaying actions by the maximum latency for all of the devices*".

Kim et al. introduced in the paper "*Inter-Device Media Synchronization in Multi-Screen Environment*" [84] a concept for synchronizing media streams across devices by exchanging playback timing information between involved devices and adjusting the system clock on each device to a common reference time from a central server. "*The actual inter-device media synchronization algorithm consists of exchanging timestamps, computing time offsets according to the round-trip delay between the server and client, and adjusting the system time*" [84].

Klos et al. discussed in the paper "*Three Challenges for Web&TV*" [85] the differences between Web Browsers and TVs and introduced a new approach for migrating functionalities of a set-top-box (STB) to the cloud. It allows operators to provide low-cost devices like HDMI dongles which only need to render media. The entire application runs in the cloud which captures and streams the UI output to the client as a video.

Howson et al. introduced in the paper "*Second screen TV synchronization*" [86] a concept for synchronizing audiovisual content delivered using different transport protocols and in different networks. The authors focus in this paper on synchronizing broadcast and broadband content across devices. The challenges addressed in the paper are the different wall clocks used in broadband and broadcast and the differences in latency for receiving content in both networks. In this context, the wall clock measures the time elapsed since the start of a TV program. The paper solves this issue by inserting an "*Event Timeline*" in the broadcast stream which contains information about the event itself and a timestamp according to a reference clock. The players on the client devices can use this information to adjust the playback.

Tolstoi el al. presented in the paper "*An Augmented Reality Multi-Device Game*" [87] a concept for using multiple devices and augmented reality techniques to play a game and increase the immersive user experience. The authors demonstrated their concept using a strategy game called "tower defense" and two devices, a tablet and smartphone. The tablet shows the main game field and allows a user interaction using touch. The player can use a mobile device to enhance the user experience by showing additional interactive elements. The position of the elements on the smartphone screen is determined using motion sensors on the smartphone. The game state is synchronized by exchanging the state of the game on each device in the local network.

Sarkis et al. introduced in the paper "*A multi-screen refactoring system for video-centric web applications*" [88] an authoring system that allows end users to split the user interface of a web application which is designed to run on a single screen and migrate part of it to other screens. The refactoring system focuses on web applications where the user interface is defined using HTML and JavaScript. The Browser stores the state of the web application in a so-called Document Object Model (DOM). The refactoring system operates on the DOM tree and based on user selection splits it into different sub-trees which can be migrated to other screens. Migration means that a copy of the selected part of the DOM tree is created and displayed on the target device and kept in sync with the original DOM on the host device. The challenge of the system is to adapt the application to the target screen by considering all input and output capabilities. Also, the split of the application designed to run on a single screen can affect the usability. Currently, there are no existing applications in production that use this or a similar concept.

A similar approach is followed by Oh et al. in their research "*A remote user interface framework for collaborative services using globally internetworked smart appliances*" [89]. They introduced a Remote User Interface (RUI) framework that allows users to mirror the entire UI or share part of it on devices in the same or different networks. Each home network consists of a gateway that connects devices in a home network to an RUI server. The RUI server is needed if devices in different networks need to connect. Devices in the local network can use SSDP as a discovery protocol. The framework also provides a virtual IO component that captures user inputs like touch on host devices and triggers them on the presentation device using appropriate events that are supported on the target device.

Jin et al. presented in the paper "*Multi-Screen Cloud Social TV: transforming TV experience into 21st century*" [90] a multiscreen cloud social TV framework which encapsulates a set of media services that can be composed together using a set of APIs and a multiscreen orchestration protocol to build social TV applications on multiple screens. This research identifies relevant functions related to social TV

experiences and offers for each of them a component with integrated UI like "Video Chat", "Text Chat", "Video Comment", and "Video Player" which can be composed together in a social TV application. Each of the components can be easily migrated between devices connected to the TV. For example, the user may move the chat component to the mobile device and keep the video player component on the TV.

Krug et al. followed in their research "*SmartComposition: A Component-Based Approach for Creating Multi-screen Mashups*" [91] a similar approach as considered in the research from Jin et al. [90] which is based on individual components that can be composed together to build a multiscreen mashup. The system is called SmartComposition and extends the Open Mashup Description Language (OMDL) developed in the EU FP7 Project OMELETTE [92]. OMDL is designed initially to build single screen mashups by composing reusable web components or widgets. SmartComposition extends OMDL to support multiscreen mashups by extending the inter-widget communication model to support multiple screens based on the publish/subscribe pattern.

Martinez-Pabon et al. proposed in their research article "*Smart TV-Smartphone Multi-screen Interactive Middleware for Public Displays*" [93] a new concept for multiscreen interaction focusing on non-personal devices like public displays. The approach followed in this article is a loosely coupled interaction model based on the publish/subscribe paradigm. It utilizes the Web Application Message Protocol (WAMP) which implements the publish/subscribe approach on top of WebSocket. An Android reference implementation is available and used to evaluate the system with an advertisement scenario on digital signage displays located in shopping malls. It is still not clear how the user can link the mobile device with the public screen. It only focuses on the messaging between different applications distributed on multiple devices.

Yoon et al. present in the research article "*Classification of N-Screen Services and its Standardization*" [94] the results of a study focusing on scenarios for consuming content on multiple terminals with different capabilities. The study aligns with the standardization activities in the ITU-T Study Group 13 (ITU-T SG13) which addresses service scenarios over FMC (Fixed-Mobile Convergence). The study classifies the scenarios in three categories: 1) deliver the same content to multiple screens with different capabilities, 2) migrate content from one device to another and 3) consume different content on multiple screens in a collaborative manner. The study describes a model based on the five entities Person, Terminal, Network, Content, and Service.

Xie et al. introduced in their research "*The design and implementation of the multi-screen interaction service architecture for the Real-Time streaming media*" [95] a method for rendering applications on a remote machine (server) hosted in the local

network or the cloud. The server consists of components for capturing the UI output of the application, encoding using MPEG-4 and streaming using the Real Time Streaming Protocol (RTSP). The client which runs on a TV or a smartphone consists of a video player and a component for sending user inputs to the remote rendering server. The presented approach applies to any application, but the authors used a game for the evaluation.

Lee et al. introduced in their research "*Remote Application Control Technology and Implementation of HTML5-based Smart TV Platform*" [96] an approach for controlling HTML5 Smart TV applications remotely based on JSON-RPC specification. The approach provides features similar to DIAL, but using other protocols. DIAL uses SSDP for discovery and REST for launching and stopping applications while the remote control protocol abstracts from the underlying discovery protocol by defining an abstract discovery API and App control protocol using JSON-RPC on top of WebSocket. However, the fundamental difference between both protocols is the location of the server which runs on the TV in case of DIAL and on the mobile device in case of the remote control protocol. Running a server on a mobile device may have implications regarding battery life, privacy, and security.

Abreu et al. followed in the research "*Enriching Second-Screen Experiences with Automatic Content Recognition*" [97] another approach for enhancing the TV viewing experience using second screen applications. The main difference to the previous work is the synchronization of content on TV and second screen. In this approach, the author used Automatic Content Recognition (ACR) techniques through audio-fingerprinting to identify the content displayed on the TV and present enhanced information on the second screen. There is no need to run a Smart TV application which enables content providers and broadcasters to provide services without relying on a specific TV platform or manufacturer. The solution works with broadcast and broadband content in the same way. The authors evaluated the introduced approach using a second screen application called "*2NDVISION*" which identifies content on the TV using audio fingerprint and image recognition technologies. One disadvantage of the ACR solutions is the lack of privacy since the second screen needs to capture audio or video from the microphone or camera and send them to a server to recognize the content.

Yoon et al. presented in the paper "*Thumbnail-based Interaction Method for Interactive Video in Multi-Screen Environment*" [98] an approach for supporting interactive video on multiple screens. The TV displays the main video and the second screen shows interactive elements related to content on the TV. The main limitation of this kind of scenarios is the user experience since there is no intuitive connection between elements in the second screen and objects appearing in the video on the TV. To solve this issue, the authors followed a new approach by creating thumbnails from specific

keyframes in the video and making them available in the second screen synchronized with the video playback on the TV. The second screen shows the interactive elements on top of the thumbnails at the same position of the object in the video on the TV.

Punt et al. focused in the paper "*Rebooting the TV-centric gaming concept for modern multiscreen Over-The-Top service*" [99] on the gaming domain. The authors introduced a framework called SHARP to develop TV-centric games and provided a proof-of-concept implementation for Android TV. The primary approach for TV-centric gaming is using the TV to show the common game field and mobile devices as game controllers or to show private information. The authors also discussed other options like displaying the game field on top of broadcast content and use information from the broadcast or OTT content like EPG in the game.

Centieiro et al. published in their research "*Enhancing Remote Spectators Experience During Live Sports Broadcasts with Second Screen Applications*" [100] a study about engaging users while watching sports events using the second screen. For this purpose, four second-screen application prototypes were developed and evaluated. The motivation behind these applications is to enhance the user engagement during a soccer game in different ways. The first application "WeApplaud" is designed to allow a group of users to participate in the applause happening in the stadium using the smartphone. The accelerometer and the microphone were used to detect a clap and clap intensity of each team will be collected and displayed in sync on the TV. Also, vibration sensors of the smartphone are used to alert users to initiate a synchronized applaud for their team. The second application of the study is called "WeBet" which prompts users to bet if a goal in a soccer game is about to happen. The innovative idea of this app is "eyes-free interaction" which allows a user to bet without to look in the smartphone by using gestures. The third application "WeFeel" allows friends to share their opinions and emotions while watching a sports event in a minimally disruptive way by using a straightforward interaction on the mobile device to express emotions and displaying emotions of friends as an overlay on the TV screen. The last application "WeSync" synchronizes the second screen application with the broadcast stream on the TV which can be delayed compared to the live event at the stadium and affect the user experience. The authors decided not to use the ACR approach to synchronize the second screen with the live broadcast and used a manual approach instead. The user needs to answer specific questions related to specific moments in the game, and the delay will be guessed. The results showed a significant engagement of users participating in the study and considerable interest in this kind of applications.

Geerts et al. investigated in their experiment "*In front of and behind the second screen: viewer and producer perspectives on a companion app*" [101] second screen applications. The authors evaluated the user experience of the second screen by

addressing issues producers face during the development and deployment of second screen applications based on interviews with producers and observations of viewers using a camera placed in their living room. The experiment focuses on the drama series "De Ridder" and its second screen application, which consists of a timeline showing information related to the current scene on the TV. The result of the experiment shows that the way how to connect the second screen to the TV must be straightforward and intuitive. For example, creating a profile and requesting the user to login on both the TV and the second screen is ambiguous. A zero-conf mechanism for discovering and pairing devices should be used instead. Another outcome of the experiment is to use a single app for all programs of a broadcaster and not a single app for each program. Live synchronization between second screen and broadcast is also crucial. Audio watermarking was not good enough for synchronization and took 6-10 seconds which is too long and decreases the user experience. Also, it is essential to offer the synchronization not only during the scheduled broadcast time but also for recorded programs. The experiment also provides recommendations for improving features related to Timing, Social Interaction, Attention, and Added Value.

Seetharamu et al. showed in the paper "*TV remote control via wearable smart watch device*" [102] how to use a smartwatch instead of a smartphone to control a TV. The smartphone is still involved since the smartwatch cannot directly communicate with the TV. The smartwatch consists of many sensors that can be used to detect user gestures and control the TV accordingly. Thereby, the smartwatch sends the sensor data to the smartphone which detects the gesture by evaluating the received data and controls the TV according to the selected rule. In most cases, the connection between the smartwatch and the smartphone is established using Bluetooth Low Energy (BLE) while the smartphone establishes a connection to the TV over the local WiFi network. The authors evaluated the solution by using the smartwatch to control the web browser running on the TV, for example, to scroll through the page or to change the active tab.

2.4.2 Multiscreen Multimedia Content

One of the fundamental concepts for the delivery and playback of multimedia content on devices with varying characteristics like screen resolution, media decoding capabilities, and available bandwidth is adaptive streaming. It enables the selection of media content that is best suited for a particular device and under certain conditions. Stockhammer et al. introduced the standardization of adaptive streaming in the paper "*Dynamic adaptive streaming over HTTP - standards and design principles*" [103]. The paper focuses on the Dynamic Adaptive Streaming over HTTP (DASH) standard which includes a specification of media presentation, formats of media

segments, and delivery protocols. It also supports different service types like Live, On-Demand, and Time-Shift. The basic principles of DASH and other HTTP-based streaming protocols such as HTTP Live Streaming (HLS) are to replace traditional stateful streaming protocols such as Real-Time Streaming Protocol (RTSP) with a stateless delivery approach based on HTTP. Before the introduction of DASH, most HTTP-based solutions were based on progressive download and using HTTP byte range requests. Progressive download has many disadvantages especially regarding wasted bandwidth and missing support of adaptive bitrate. DASH addresses these issues by breaking down the media content into short media segments which can be delivered over HTTP and played independently from each other. DASH also considers the generation of multiple versions of the media segments using different bitrates and media codecs. The XML-based Media Presentation Description (MPD) provides information about media segments and other metadata the client needs to request and play the content. The DASH segments and the MPD can be hosted on an HTTP server while the entire logic for streaming and playback is implemented in the player. Existing Content Delivery Networks (CDNs) which are successfully used for delivering web pages and static web content can be also used as a scalable and reliable system for delivering DASH content.

Niamut et al. introduced in the paper "*MPEG DASH SRD: Spatial Relationship Description*" [104] an extension of the DASH protocol to support spatial media. The Spatial Relationship Description (SRD) feature of DASH enables the streaming of parts of a video in different qualities. The SRD extends the MPD to describe the relationships between associated parts of video content. It allows a *client to select and retrieve only those video streams at those resolutions that are relevant to the user experience*. There are multiple application scenarios where SRD can be applied like "High-quality zoom-in" where the viewer can zoom-in in a UHD video with the best available quality for each zoom level. The client uses the SRD information from the MPD and selects only the video segments for the requested region of interest in the appropriate bitrate.

Jung et al. introduced in their research "*A web-based media synchronization framework for MPEG-DASH*" [105] a peer-to-peer based mechanism to synchronize DASH content on multiple screens using web technologies. The framework synchronizes audio and video content across different devices using WebRTC [54] for exchanging the playback states across devices and the Media Source Extension API (MSE) [63] for playing back DASH content in the Browser.

Another application domain for DASH SRD is the streaming of spherical 360° videos. Hosseini et al. focused in their research "*Adaptive 360 VR Video Streaming Based on MPEG-DASH SRD*" [106] on this domain by applying the SRD concept to spherical videos in order to reduce the required bandwidth by streaming high-resolution

content like 8K and 12K. The basic idea of the presented solution is that the viewer can only see a fraction of the 360° video at a particular time, while the other part remains unseen. The 360° video will be spatially divided into several tiles, and each of them will be encoded in different bitrates according to the SRD concept. The 360° player will request the tiles in the bitrate according to the SRD metadata from the MPD file. Tiles inside the viewport will be requested with the highest reasonable bitrate while tiles outside the viewport will be requested in a lower bitrate. The authors did not mention the supported video codecs and the costs for merging the tiles.

Concolato et al. proposed in their research "*Adaptive Streaming of HEVC Tiled Videos using MPEG-DASH*" [107] a tile-based approach using DASH SRD together with High-Efficiency Video Coding (HEVC) [58]. The paper describes the whole process from content preparation where the source content is split into different tiles, and those are packaged as ISOBMFF/HEVC compliant video segments that can be referenced independently by the client. ISOBMFF stands for ISO Base Media File Format [62] which is a structural, codec-independent file format. On the client side, an HEVC compliant single stream is created from selected tiles which are described in the DASH MPD. The paper deals also with the challenges related to HEVC encoding, storage of HEVC tiles in ISOBMFF format and DASH content generation.

Van Brandenburg et al. and Niamut et al. focused in their research "*Spatial segmentation for immersive media delivery*" [108] and "*Towards A Format-agnostic Approach for Production, Delivery and Rendering of Immersive Media*" [109] on an approach similar to [106] and introduced a system which is able to capture, create and deliver immersive videos where users can interact with the content via Pan, Tilt or Zoom (PTZ). Only Ultra High-resolution panorama videos are considered in these works. Spatial segmentation is used as a method to efficiently deliver parts of an ultra high-resolution video to devices which are not capable of displaying the entire resolution at once. In order to save process resources for the decoding and recombination of several spatial segments on target devices, these tasks are performed in the cloud on so-called Segment Recombination Nodes (SRN).

Mavlankar et al. introduced in their research "*Peer-to-peer multicast live video streaming with interactive virtual pan/tilt/zoom functionality*" [110] an approach based on P2P multicast for delivering video with interactive region-of-interest (IROI) to peers. The application scenario motivated by this work is the same as in the previous research mentioned in [108] but following a different approach for delivering the video segments using a P2P Overlay Protocol. The new protocol tracks the available video segments on each peer to deliver content between peers in the networks based on their location and available videos without the need to stream the content from the source server.

Wen et al. introduced in the research "*Cloud Mobile Media: Reflections and Outlook*" [111] a media platform focused on one fundamental principle to reduce the capability requirement on content sources and playback devices. The system introduced in this work is based on cloud computing paradigms which enable the migration of resource-intensive tasks like media transcoding and caching into the cloud. Zare et al. also follow a similar approach in their research "*HEVC-compliant Tile-based Streaming of Panoramic Video for Virtual Reality Applications*" [112] with focus on 360° videos.

Jin et al. proposed in their research "*Reducing Operational Costs in Cloud Social TV: An Opportunity for Cloud Cloning*" [113] another approach that uses cloud computing paradigms for media delivery and playback in a multiscreen environment. The main idea of the proposed approach is to instantiate a virtual machine in the cloud called "cloud clone" for each user. The cloud clone provides essential features like application execution, media transcoding, ad insertion, and session management. The cloud clone allows users to migrate sessions between devices and to mirror the application running in the cloud clone to multiple devices. The research focuses mainly on reducing the cost for such a deployment by finding the best location in the network for the cloud clone. The authors formulated the problem as a Markov Decision Process to balance a trade-off between the transmission and migration costs. It is not clear from the results how the system performs in a large scale deployment.

Carlsson et al. presented in their research "*Optimized Adaptive Streaming of Multi-video Stream Bundles*" [114] an approach for the delivery and synchronized playback of multi-view videos. The authors introduce a concept of “multi-video stream bundle” which includes the different video streams for all camera views and deliver the whole content to the client using adaptive streaming techniques. The challenge is to reduce the required bandwidth since only one camera view is shown at a specific time while the other views remain unseen until the user manually changes the view. A core element of the system is the content prefetching and buffer management component which applies an optimization model based on heuristics to balance the playback quality and the probability of playback interruptions. This approach streams the current camera view in a quality that consumes a specific amount of the available bandwidth, and the remaining bandwidth will be used to prefetch other camera views in lower qualities.

Gunkel et al. introduced in their research "*WebVR meets WebRTC: Towards 360-degree social VR experiences*" [115] a VR framework that addresses the problem of isolating users while watching 360° videos on head-mounted displays. The framework is based on the web technologies WebVR [73] and WebRTC [54] and extends existing video conferencing systems with new virtual reality functionalities. For example,

multiple users can watch TV or play games together via synchronized playout. The WebRTC connection is used to synchronize the playback or game state and share content in a peer-to-peer manner.

Belleman et al. discussed in the paper "*Immersive Virtual Reality on commodity hardware*" [116] new approaches to build and run VR systems on low-end devices. The research was more focused on enterprise and scientific VR applications, for example, to explore data from live, large-scale simulations. At the time of publication, this type of application required special VR systems that were only available in research labs. The paper identified the need to run VR applications on commodity hardware and showed in the first experiments acceptable performance for rendering VR content on a PC and displaying the output on a connected VR system.

Qian et al. presented in the "*Optimizing 360 video delivery over cellular networks*" [117] an approach that addresses the critical aspects of 360° video playback such as performance and resource consumption. The approach followed in the paper considers an *infrastructure that facilitates ubiquitous access of VR resources in the cloud*. The authors proposed a cellular friendly streaming mechanism for 360° videos which only fetches the visible part of the video instead of downloading the whole content in order to reduce the bandwidth consumption. A part of the solution is a component that predicts the head movement of the user in order to prefetch content for the predicted FOV in advance. The accuracy of the prediction is evaluated in an experiment that resulted in an accuracy of 90% .

Neng et al. introduced in the paper "*Get around 360° hypervideo*" [118] a concept for enabling interactions in 360° videos through clickable objects and customizable overlays. The authors propose an interactive layer on top of the 360° video which includes an indicator where the user can see in which direction he is looking and a mini-map that contains thumbnails of the original video (equirectangular) with a visualization of the interactive spots in the visible and non-visible area. If the user selects an object which is not in the current FOV, the virtual camera moves to the position of the selected element in the 360° video. The focus in the paper was only on touchscreen and desktop devices.

Pang et al. introduced in the paper "*Mobile interactive region-of-interest video streaming with crowd-driven prefetching*" [119] an approach for interactively selecting region-of-interests (ROIs) in a video by using Pan/Tilt/Zoom (PTZ) controls while saving bandwidth. The provided solution facilitates displaying the selected ROI in the best possible quality without wasting bandwidth for downloading unseen ROIs. At the same time, the paper provides a solution for low-latency switching between ROIs using a crowd-driven prediction scheme to prefetch regions that are expected to be selected next. The authors focus in the paper on mobile devices like tablets and

smartphones where PTZ controls can be easily implemented using touch inputs, but the concept can be applied to any playback device by adapting to the PTZ controls to the input methods provided on these devices. Similar to other approaches, this solution splits the video in different tiles that can be encoded independently using H264/AVC. Since most clients are only capable of decoding one video at a time, the solution provides a mechanism to decode a ROI in a single tile, but the tiles may overlap and can have different dimensions or zoom levels. The system can automatically produce ROI videos that track objects of interest, and only these videos are available to the viewer.

Ochi et al. introduced in the paper "*HMD Viewing Spherical Video Streaming System*" [120] an approach to reduce the bandwidth required to stream 360° videos by generating multiple versions of the video where certain regions are encoded at a higher bitrate while the remaining regions at a lower bitrate. The player requests the video version with the highest overlap between the high bitrate region and the current FOV. When the user changes the FOV, the player requests the new video version corresponding to the new FOV. The player may show the new FOV in lower quality after changing the view until the video segments of the new FOV are loaded.

2.5 Discussion

In this chapter we have discussed relevant state-of-the-art solutions and related works in the field of multimedia multiscreen applications and content. We have shown on the one hand the relevance of this research area, but on the other hand that a uniform concept for modeling multiscreen applications and the possibility to implement them in a standardized way, especially on the Web, is still missing. Most related work focuses on providing customized solutions for specific multiscreen features, but not on how these solutions, in combination with standard APIs and protocols, can be used to make multiscreen application development as easy as for single-screen applications by hiding the complexity of the underlying components and technologies. [74] [75] and [76] introduce similar approaches that allow Web pages to discover and connect to home networked devices using specific technologies like UPnP and mDNS without providing any concept or model for building multiscreen applications. It has become evident that these approaches bring security and privacy risks, as a Web page gets a direct access to critical services in the home network and for this reason the work on the API in [76] has been discontinued. [78] introduces a framework for supporting multiscreen Web applications using a cloud rendering approach which has limitations regarding offline support, access to device APIs as well as video and graphical rendering capabilities. This thesis will abstract from

the underlying runtime environment by providing a new concept for modeling multiscreen applications (Section 4.2) that can be applied to different runtime architectures (Section 4.4.1) including cloud rendering. [79] and [80] follow a different approach as in [78] by distributing the application execution on multiple devices in the home network. This approach has some drawbacks if one or more devices are not able to execute the parts of the application assigned to them, e.g., due to limitations on graphical processing, computation or media rendering capabilities on these devices. This thesis addresses this issue by allowing application components to be rendered remotely on other devices that are able to execute these components or in the cloud without the need to update the application. [83] uses social media networks as an underlying platform for discovering and connecting devices. The approach presented has a strong dependency on third-party services that can store critical application data and increase the risk of data misuse. This thesis follows *Separation of Concerns* design principles using modular and reusable components that work across multiple runtime architectures (Section 4.4) and allow switching between them at minimal cost. This also applies to the approach presented in [85], which uses cloud rendering mechanisms to support low-cost TV dongles that are powerless to render the application locally. [95] follows also a similar approach. [88] presents a refactoring system that makes it possible to split a Web page and transfer parts of it to other screens with almost no additional costs. This approach is well suited for simple video-based applications, but is difficult to use for complex applications that are not designed to run on multiple screens. [89] follows also a similar approach. This thesis provides the tools and concepts for designing, modeling, and implementing multiscreen applications while reducing development costs and time through a number of software components (Section 4.4.2) that are used in almost every multiscreen application. [87] and [99] show the importance of multiplayer games using multiple screens which is also considered in this thesis (Section 3.1.2) in order to derive the functional and non-functional requirements especially regarding the synchronization of game state across different devices using state-of-the-art synchronization techniques (Section 4.3.3). [100] and [98] show the relevance of using second screens as companion devices for broadcast services running on the TV. This thesis also deals with this type of scenarios (Section 3.1.4) and considers their requirements. [90] introduces a set of social TV application components that can be freely moved between devices especially between TV and smartphone. This thesis expands this approach by introducing modular and reusable components (Section 4.2) that work across different domains and are not limited to a specific one. [91] extends the concept of *Mashups* to support multiple screens using modular components called widgets, and by using an event-based approach for the interaction between these widgets. One limitation of the mashup approach is the way how several widgets can share a single screen. A widget can occupy a part of the screen and can be exclusively used by it. This thesis presents two different types of application components, *atomic* and *composite* (Section 4.1). Composite

components allow developers to define how atomic components running on the same device can share common resources. In addition to the event-driven approach, this thesis also introduces the message-driven and data-driven approaches (Section 4.3), which can be individually selected by the application developer. [101] addresses issues producers are facing during the development and deployment of second screen applications. This thesis addresses the findings presented on this study regarding the usage of simple mechanisms to discover and connect to devices with minimal interaction with the user via network service discovery techniques or a new approach using iBeacons (Section 4.6.1). Another finding of the study is the poor user experience of using *Automatic Content Recognition* techniques for synchronizing content across devices. This thesis also addresses this finding by using dedicated communication channels for exchanging playback and timing information between the screens (Section 5.2.2).

We have also introduced in this chapter state-of-the-art technologies and research activities for the delivery and playback of multimedia content in a multiscreen environment such as Adaptive Bitrate Streaming, Content Delivery Networks, Tiled Media and Web APIs that enable the efficient delivery and playback of multimedia content on almost any device and platform. However, the introduction of new multimedia formats, such as 360° video, poses new challenges on the bandwidth and graphical processing capabilities of target devices. This results in a limitation of the number of supported devices and platforms. This thesis will address these new requirements and present a new approach for the playback of 360° video on embedded devices with limited capabilities and bandwidth. [104] introduces an extension of MPEG-DASH called *Spatial Relationship Description* which is considered in this thesis to describe tiled video content for synchronized and adaptive playback in a video wall (Section 5.2). [105] already presents an algorithm for synchronizing DASH content on multiple screens using WebRTC. The introduced algorithm assumes that the latency for the communication between the master and slave clients is constant or insignificant and the browsers used on all screens are from the same vendor. This thesis extends this algorithm by considering the offset between the clocks on the different screens using state-of-the-art time synchronization techniques (Section 5.2.2). The new algorithm introduced in this thesis also keeps the quality levels (DASH representations) of all video tiles displayed on the different screens constant by monitoring the amount of buffered content and the bandwidth on each individual display for calculating the best appropriate quality level at a specific time. The proof of concept implementation of the synchronization algorithm using HTML5 video element considers the inaccuracy of using the seeking method to adjust the current playback position. Instead, the algorithm introduced in this thesis uses a more accuate method by updating the playback rate of the video to adjust the playback position on the individual displays. [106], [108] and [109] present similar approaches of using DASH and HEVC tiled streaming for efficient delivery

and playback of immersive content. The main limitation of these and other similar approaches like [120] that rely on client-side transformation (Section 5.3.2) is the limited support of low-capability devices like TVs and low-cost streaming devices that are not able to render the *Field of View* or *Region of Interest* locally. [117] and [112] present new approaches that rely on server-side transformation (Section 5.3.2) by moving processing intensive tasks like FOV rendering to the cloud. These approaches are not suitable for media delivery for the mass audience due to the operation costs and scalability limitations. This thesis introduces a new approach that tackles the limitations of these approaches based on the pre-rendering of FOV videos (5.3.4).

3 Use Cases and Requirements Analysis

This chapter defines and discusses a number of relevant use cases in the field of multi-screen multimedia applications, intended to form the foundation for deriving and analysing functional and non-functional requirements. These requirements will be considered and evaluated in Chapters 4 and 5 to develop and implement a concept for creating multiscreen applications and multimedia content. The remainder of this chapter is structured as follows: Section 3.1 defines and describes the use cases while Section 3.2 focuses on identifying and analyzing the requirements from the use cases.

3.1 Use Cases

There are many use cases for consuming media content, playing games, displaying information and running other types of applications on multiple screens such as TVs, smartphones, tablets, and PCs. However, there are also several aspects when it comes to designing and developing such applications that are common to all of these areas. Finding out these aspects will guide us in developing concepts and models for building applications across different domains and identify the building blocks for the underlying software components that can be reused across different application areas. It is essential to define the use cases that cover all potential real-world scenarios in each domain. These use cases are described in the following sections. Also, each use case is assigned a list of related real-word examples as evidence of its relevance in this context.

3.1.1 UC1: Remote Media Playback

Remote Media Playback is one of the most important and popular use cases in the multiscreen domain. It is supported in different variations by most popular Video and Music Streaming services like YouTube [4] and Netflix [24]. These services may support different types of remote playback devices, but the basic flow is always the same which is illustrated in Figure 3.1 and described in the steps below:

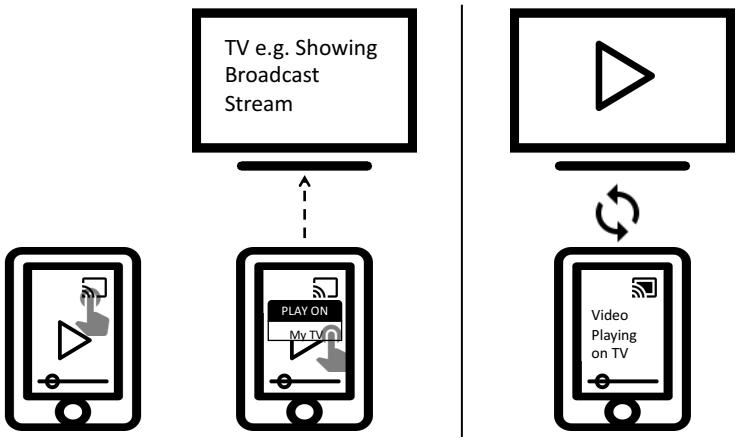


Figure 3.1.: UC1: Remote Media Playback

1. As a first step, the user starts the video or music application on a mobile device. The application can be downloaded from an App Store and installed on the device, or it can be a simple web page that can be opened in the browser after entering the service URL.

2. The user browses the media catalog in the application looking for specific content and starts the playback on his local device after selecting a specific media item. The application offers UI elements which allow the user to control the media playback like play, pause, and seek.

3. In the mobile application there may also be a button indicating that there is a remote playback device available, e.g., a TV. Its state indicates that the application is currently not connected to a remote playback device. The user decides to continue watching the video on the TV. He clicks on the button and selects a remote playback device from the list.

4. Now the playback stops on the local device and continues from the same position on the TV. The local player shows a message that the video is currently playing on the TV. The user can still control the video from the mobile application. Furthermore, the current time and playback state are always synchronized between local and remote playback devices.

5. The user can disconnect at any time from the remote playback device, and the video playback continues on the mobile device from the current position. The user can also disconnect from the remote device without terminating the video playback and can reconnect to it at a later time.

In addition to the basic flow of the Remote Media Playback use case, there are important aspects to discuss that can lead to requirements which cannot be derived

from the steps described above. For example, the application must ensure that the remote playback device can play the selected video. Otherwise, the user will receive an error after starting the video on the remote device which will affect the user experience. Another relevant aspect is the ability to customize the remote player, i.e., to display additional content such as advertisements over the video, which is essential for most commercial applications. Furthermore, it is crucial to provide the video in the quality that fits best to the screen size of the playback device.

3.1.2 UC2: Multiscreen Game

Gaming is another important category of applications that benefit from multiple screens to improve the user experience while playing a game. Many popular games like Angry Birds [121] already support "multiple screen" mode. There are many variations on how the different screens can be used in a game. The simplest case is to use the smartphone or tablet as a replacement for the physical controller of a game console. In other situations, a large screen like a TV can be used to extend the view of the game field. In two-player or multi-player games, a third screen can be used to display the common game field while each personal screen displays the player UI. The following steps illustrate the flow for all these variations using a card game showed in Figure 3.2 as an example:

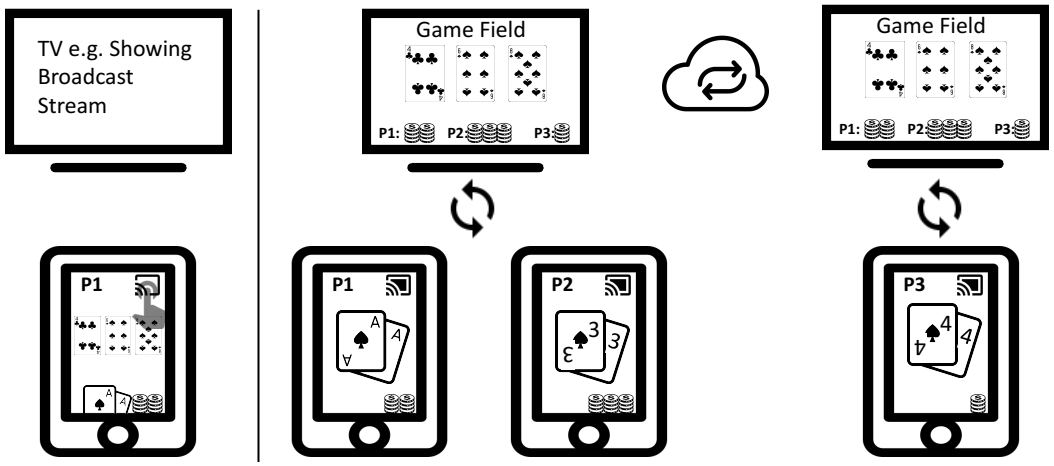


Figure 3.2.: UC2: Multiscreen Game

1. A Player "P1" launches a card game on his smartphone or tablet in single player mode and starts playing a game against the computer. The game field consists of two parts, the first one shows the player's private cards and coins while the second part is the common game field that shows common game information among all players like open cards and game state.

2. A second player "P2" decides to participate in the game. Since the two players "P1" and "P2" are in the same physical place, they decide to switch to the

multiscreen mode, in which each player can use his device while the common playing field is displayed on the TV. For this, the player "P1" clicks on a button in the application to connect to the TV. As soon as the connection is established and the game has started on the TV, the common game field moves from the screen of player "P1" to the TV.

3. Now player "P2" can participate in the game. He starts the application as for player "P1" and clicks on the same button to connect to the TV. Player "P2" can now choose whether he wants to start a new game or join the already running game. He chooses the option to join the game.

4. Player "P1" decides to invite a third player "P3", who is not at the same place, to the game. For this, he creates and sends an invitation link. After player "P3" has received the invitation, he clicks on the link to start the game and participates in the current session. Player "P3" can also connect and migrate the common game field on his TV at home.

The final state of this use case is depicted in the second part of Figure 3.2 where five screens in two different places are involved. The game flow described above can be applied to nearly any multiscreen game, but in some situations, there are additional aspects that need to be considered. For example, in games where intensive graphical processing is required, it is essential to know if the remote device can perform the graphical processing or not. Another aspect is the latency of the interaction with the game which can affect the user experience.

3.1.3 UC3: Personalized Audio Streams

This use case shown in Figure 3.3 focuses on the synchronized playback of audio and video streams on multiple screens. It is relevant, for example, if two viewers are watching TV in the same physical location, but each of them wants to select a different audio language for the broadcast content displayed on the TV. Another important case is, if one of the viewers is visually impaired and wants to select a narration track with audio description while others can select the default audio track without audio description. These scenarios are already discussed and considered in new standards like HbbTV. The following steps illustrate the procedure of this use case.

1. Two viewers are watching a movie on a broadcast channel on the TV. The channel offers the three languages German, English and French for the audio while German is the default language delivered in the broadcast stream.

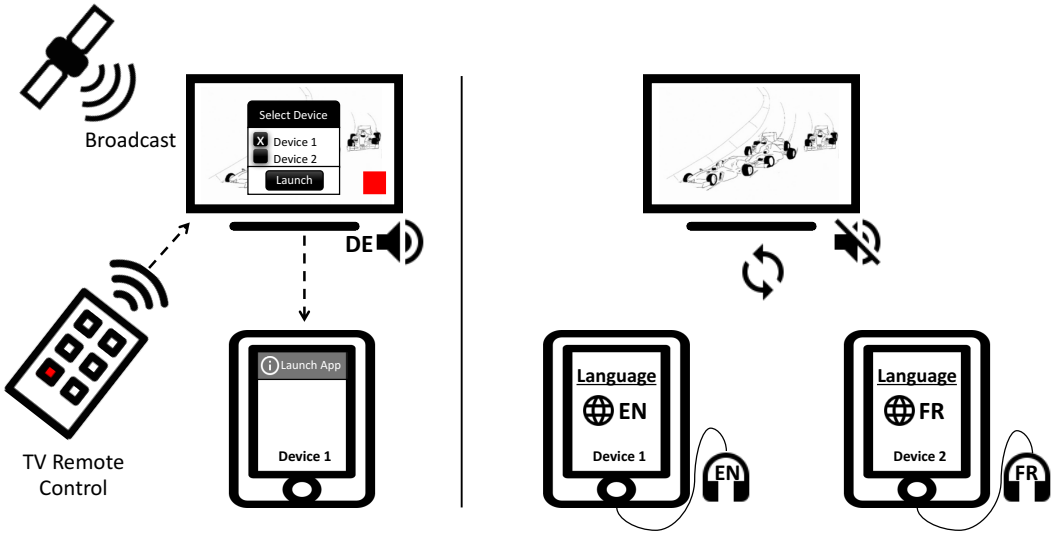


Figure 3.3.: UC3: Personalized Audio Streams

2. The first viewer decides to select the English audio track and the second viewer the French audio track. Since it is not possible to select two different audio tracks on the TV, each viewer can use his smartphone as a playback device. The audio tracks need to be always synchronized with the broadcast stream on the TV.

3. The broadcaster offers a hybrid application that runs on the TV and offers a feature to change the audio language. The first viewer selects using the TV remote control the "Audio Language" menu which shows the companion devices of the two viewers. The viewer selects his device from the list and clicks on the launch button. He receives a push notification which allows him to launch the Broadcaster's companion application on his smartphone.

4. After the broadcaster's companion application is launched, it connects automatically to the TV. The viewer can now select the English audio track. In order to not disturb the second viewer, he uses a headset instead of the loudspeakers of the smartphone.

5. The second viewer repeats the same steps, but he selects the French audio track instead. Also, he chooses the option to mute the broadcast audio on the TV as both viewers now use their smartphones as playback devices for the audio tracks.

This use case can be also applied to other audio tracks like the audio description for visually impaired viewers. The technical challenges for synchronizing broadcast video with broadband audio on different devices remain the same. There are also other scenarios for synchronizing broadcast video on TV with broadband video on

mobile devices with the same challenges. One of these application scenarios are events with multiple camera perspectives. In this case, the TV shows the main broadcast stream while the viewer can select a specific camera in the companion application.

3.1.4 UC4: Multiscreen Advertisement

The basic idea of this use case is to engage the viewer with the content displayed on the TV. Multiscreen Advertisement is a very important commercial use case for many broadcasters. The basic flow of this use case is depicted in Figure 3.4 and described below:

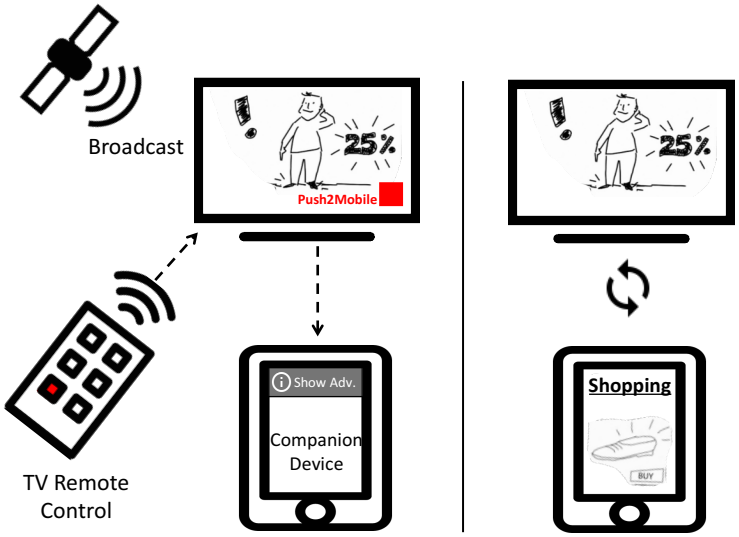


Figure 3.4.: UC4: Multiscreen Advertisement

1. A viewer is watching a broadcast channel on the TV. During the Ad break, the TV shows the option "Push2Mobile" which allows the viewer to launch a companion application with just one interaction, i.e., via the red button of the TV remote control as depicted in the first part of Figure 3.4.

2. The viewer presses the red button to get more information about the advertisement displayed on the TV and receives a notification on his smartphone. The viewer will not be asked to select a specific companion device that should receive the notification. Instead, all viewer devices already paired with the TV will receive the notification.

3. The viewer clicks on the notification and a web page related to the TV advertisement opens in the browser. The companion page connects automatically to the TV and shows information related to the advertisement, for example, details about the product and how to buy it online or in a store nearby.

4. The viewer can configure the TV app to send always a notification when a TV advertisement from a specific category starts on the TV without the need to press the red button.

There are also other non-commercial use cases with the same technical challenges. For example, the companion app can show additional information about actors in the current TV show or display related content from social media networks. The primary challenge in all these cases is how to discover only companion devices of viewers sitting in front of the TV and not any device paired with the TV but currently not in the same place.

3.1.5 UC5: Tiled Media Playback on Multiple Displays

This use case is about using multiple displays in a specific order, i.e., in a matrix form to build a larger presentation screen. This kind of installations is popular for public screens. Traditional installations use specific hardware that connects all displays and exposes them to applications as a single virtual display. The underlying system will then map each part of the virtual display to its corresponding physical display. The use case depicted in Figure 3.5 provides an alternative method to present content on multiple displays using multiscreen technologies without the need for additional hardware. The basic idea is to split the content to multiple tiles in advance, and each of these tiles will be assigned to a specific display. The flow of this use case is described below:

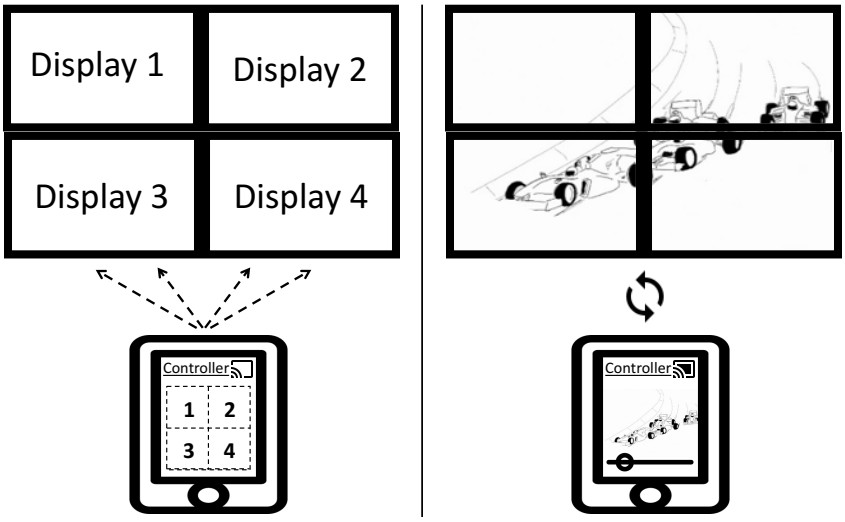


Figure 3.5.: UC5: Tiled Media Playback on Multiple Displays

1. An organizer needs to present video content on a public screen for a large audience during an event. The video is available in UHD (Ultra HD) with a

resolution of 3840x2160 pixels which has four times more pixels than FHD (Full HD) video which has a resolution of 1920x1080 pixels.

2. The organizer decides to use an installation of four FHD displays in a setup of a 2x2 matrix and form a larger virtual display which can present UHD videos.

3. In order to create content for each display, it is necessary to split the video into four tiles which can be delivered and played back on the individual displays.

4. In order to select a video and control the playback, a control application running on a tablet as depicted on the left side of Figure 3.5 is used. It launches the video application with the corresponding tile on each display.

5. The displays play back the video tiles in sync with the video playing in the control application running on the tablet which also allows the user to control the video playback or stop the video on all displays at any time.

The biggest challenge in this application scenario is the frame-accurate synchronization of the video tiles on the individual displays.

3.1.6 UC6: Multiscreen 360° Video Playback

360° video is one of the new media formats that started to reach a wider audience in recent years. There is already a wide range of 360° cameras that make it possible to produce 360° videos for everyone not just for professionals. 360° video can be played on different devices like smartphones, head-mounted displays (HMD) and TVs. This use case introduces the basic flow for watching 360° videos on different devices as depicted in Figure 3.6 and described in the steps below:

1. A user opens on his tablet a video application that supports 360° video playback. He selects a 360° video from the catalog, and the player starts rendering the field of view (FOV) from a specific angle ("View 1" depicted in Figure 3.6). Other parts of the video remain unseen. The user can change the FOV or zoom in the video using the touchscreen ("View 2" depicted in Figure 3.6).

2. The user decides to continue the video on a head-mounted display (HMD). For this, he opens the 360° video application on his smartphone which detects the last session on the tablet and offers the user to continue the video. The user connects the smartphone to the HMD, and the playback continues in VR stereo mode in which the player splits the screen into two parts for the left and right eyes. The FOV is detected using motion sensors of the HMD.

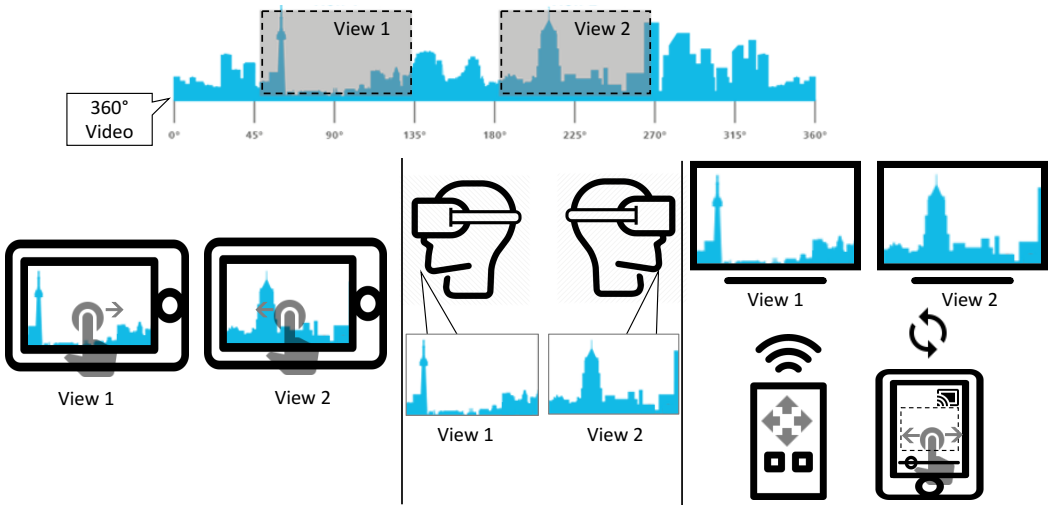


Figure 3.6.: UC6: Multiscreen 360° Video Playback

3. After a few minutes wearing the HMD, the user feels isolated and decides to continue the 360° video on the TV. After disconnecting the smartphone from the HMD, the video application discovers a TV able to play the 360° video. After establishing the connection, the 360° video playback continues on the TV. The user can use his smartphone or the TV remote control to navigate in the 360° video.

There are also other relevant features which are not addressed in this use case like live streaming and interactive 360° videos.

3.2 Requirements Analysis

This section analyses the use cases defined in the previous section and derives the functional and non-functional requirements described in the following subsections. The requirements for the application model and the underlying multiscreen system are considered in Chapter 4, while the remaining requirements for media processing, delivery and playback are studied in Chapter 5.

3.2.1 Functional Requirements

The functional requirements define the behavior and functions of multiscreen applications, media processing tools and the underlying runtime environment. Each functional requirement is described in the following subsections.

F-REQ1: Discovery

Discovery is an important requirement that enhances the user experience in a multiscreen environment. A component of a multiscreen application running on one device like a smartphone should be able to discover other devices like TVs that can display a specific content type and fulfill certain criteria without the need for additional user interaction or configuration. Some discovery technologies use the term "zero-configuration" as an indication that no additional configurations are required. The target content requested to display on the remote device could be a video, audio, image or any digital content. Optional criteria can be used for further filtering. For example, in the case of media content, it can filter devices that support a specific video or audio codec. Other filtering options related to software and hardware capabilities of the target device can also be considered. The output of the discovery step is a list of devices with all the information needed to connect to each of them and to display the requested content. The discovery should avoid finding devices that cannot display the requested content or not fulfill the input criteria. If this happens, the request for displaying the content in the next step will fail which will impact the user experience.
Related use cases: UC1, UC2, UC5, UC6

F-REQ2: Pairing

There are situations where discovery is technically not possible or has limitations. For example, a TV or any presentation device connected to the local network should only be discovered from devices in the same network. Also, personal devices like smartphones forbid to run services in the background that make the device discoverable by other devices due to security, privacy, and battery life considerations. In these situations, a manual pairing can help. It allows two devices to connect with each other for a limited or unlimited time and requires in most cases additional interactions with the user, for example, to enter a PIN-code or to scan a QR-code.
Related use cases: UC3, UC4

F-REQ3: Launch

A component of a multiscreen application running on one device should be able to launch another application component or display media content on a previously discovered device. Information of the discovered device returned after discovery or a pairing step will be used to establish a connection between sender and receiver

devices. "Sender" is the term used for devices or applications that request the launch while the "receiver" represents presentation displays on which the requested application will be launched. Some devices support only pre-installed services for displaying media content like DLNA [122] certified devices with UPnP [38] media rendering capabilities. Other device categories may support only rendering of web content in a browser installed on the device. In this case, the HTML video and audio elements can be used to render media content. A new generation of presentation devices enables the launch of any native application already installed on the receiver device. It should also be possible to launch applications or display media on multiple receiver devices simultaneously, sent from the same sender device.
Related use cases: UC1, UC2, UC5, UC6

F-REQ4: Wake-up

The remote launch of applications is not always supported especially on devices like smartphones and tablets in order to not affect the user experience. The launch of applications on these devices is only allowed after user confirmation. The only option to interact with the user while the application is not running in the foreground is through notifications. The application itself can trigger a notification if it is running in the background or by sending push notifications to the device. Triggering the application to run in the background is called wake-up. Only if the user taps on the notification, the desired application will be launched in the foreground.
Related use cases: UC3, UC4

F-REQ5: Joining

A component of a multiscreen application running on one device should be able to join a previously launched application on another device. This is important, for example, in multi-user multiscreen applications like multi-player games where one user starts the application on the receiver device while another application component running on a different device connects to it without launching a new application. Joining can also be used if a sender application launches a receiver application on a presentation device and then disconnects without terminating the receiver application. The sender application can reconnect at a later time to the receiver application using the "joining" feature.
Related use cases: UC1, UC2, UC4

F-REQ6: Terminating

A component of a multiscreen application should be able to terminate other application components previously launched from the same application. All connections to the terminated application components should be closed, and all affected application components should be notified accordingly.
Related use cases: UC1, UC2, UC5, UC6

F-REQ7: Communication

Two components of a multiscreen application running on two different devices should be able to establish bidirectional communication channels. These communication channels should support the exchange of text and binary data as well as low latency streaming of continuous binary content especially media streams.
Related use cases: UC1, UC2, UC5, UC6

F-REQ8: Synchronization

Components of a multiscreen application distributed on multiple devices should be able to synchronize their internal application state in real-time. It should also be possible to provide a frame-accurate synchronization of media streams and timed data across multiple devices. Examples of timed data are subtitles and timed captions. The synchronization can also be realized on the application level using the communication channels but the time accuracy cannot be guaranteed as it is supported at the platform level.
Related use cases: UC1, UC2, UC5, UC6

F-REQ9: Migration

An application running on one device should be able to migrate one or more of its components to another application running on a different device without losing its state. This process is called push migration. It should also be possible to migrate one or more components from a remote device to the application running on the local device which is called pull migration. Migrated components disappear from the source application and appear in the target application after completing the migration process.
Related use cases: UC1, UC2, UC6

F-REQ10: Cloning

An application should be able to clone one or more of its components to an application running on another device by keeping the state of the original and cloned components in sync. The user interfaces of the original and cloned components do not necessarily have to look the same since they can adapt to the devices they are running on.
Related use cases: UC2

F-REQ11: Instantiation

An application should be able to create a new instance of a component either locally or on a different device. Furthermore, it should be able to set the initial state of the new instance or to use the last known state of the original component before instantiation. After instantiation, the two instances of the application components should run independently from each other.
Related use cases: UC2, UC3, UC5

F-REQ12: Adaptation

An application component should adapt to the target device on which it is running. There are multiple factors related to the capabilities of the target device that should be considered during adaptation. The most important capabilities relevant for multiscreen applications are screen resolution, supported input methods and media rendering capabilities. For example, an application running on a smartphone and later migrated to a TV should adapt to the screen size and the input method (remote control) of the TV. Adaptation is not only important for applications but also for media rendering, especially video. For example, if a low-resolution video gets migrated from a smartphone to the TV, a better resolution version with appropriate encoding should be selected.
Related use cases: UC1, UC2, UC3, UC4, UC5, UC6

F-REQ13: Partial and 360° Video Rendering

We assume in this thesis that the devices should at least support regular video rendering. Also, it should be possible to render a rectangular region of a video to enable partial media rendering. In case of 360° videos, it should be possible to

display a field of view from a source video.
Related use cases: UC1, UC5, UC6

F-REQ14: Remote Media Control

A component of a multiscreen application should be able to control the playback of media content displayed on a remote device and receive the accurate playback position. In the case of partial video playback, it should also be possible to change the visible rectangular region. In 360° videos, it should be possible to change the angle and the zoom level of the field of view.
Related use cases: UC1, UC5, UC6

3.2.2 Non-Functional Requirements

In addition to the functional requirements defined in the previous section, this section focuses on the non-functional requirements which have impact on the technical implementation of the system components and the quality of service (QoS).

NF-REQ1: Motion-to-Photon Latency

Motion-to-Photon Latency is a relevant metric in multiscreen applications and multimedia rendering which has a direct impact on the user experience. It is defined as the time until a user interaction is fully reflected on the presentation screen and is widely used in gaming or 360° video applications. The maximum allowed value for Motion-to-Photon Latency on head-mounted displays is 20ms to avoid motion sickness. In multiscreen applications, the user interacts in most cases with one device while the content is displayed on another device. This means that the latency for the communication between two application components running on two different devices should also be considered in the overall Motion-to-Photon Latency. The maximum allowed value for Motion-to-Photon Latency depends on the use case. For example in case of 360° video rendering on TV and navigation via remote control or second screen, the Motion-to-Photon latency can be higher than 20ms.

NF-REQ2: Bandwidth

Bandwidth is another critical factor for multiscreen applications and multimedia rendering. It has a direct impact on the user experience, especially on the quality

of the video content that can be streamed with a specific bandwidth. It plays a more critical role in partial media and 360° video rendering where only a part of the content is displayed to the user. The challenge is to stream the content to the user device without wasting the bandwidth for streaming unseen content. In case of application rendering on remote devices, there are different requirements on bandwidth depending on how and where the application UI is rendered. For example, if the application is rendered on a remote device and the UI output is captured as a video and streamed to the presentation device, then a higher bandwidth is required compared to the case when the application UI is rendered locally on the same device where it is running.

NF-REQ3: Processing Resources

Application and media rendering requires computation and graphical processing resources, which can vary between different application scenarios. For example, some video codecs require minimum hardware capabilities in order to decode and render the video content. The rendering of 360° videos, 3D content, and graphical animations require more processing resources. The challenge is to support this kind of computing-intensive applications even on low-capability devices. Rendering 360° videos on TV is just one example.

NF-REQ4: Storage

The storage requirement addresses multimedia content rather than data and assets of the application itself. As mentioned in the introduction section, it is expected that in 2021 more than 80% of internet traffic will be video streaming which needs to be stored and cached in the network. Furthermore, in order to support adaptive streaming, different versions of the same video with different bitrates need to be created and stored or cached in the network. Creating videos with specific bitrates for individual users produces more costs than storing multiple versions of the video. The challenge is to find a balance between storage, processing and streaming requirements in order to minimize the total costs.

NF-REQ5: Scalability

Scalability is another important aspect for the deployment of a system that supports multiscreen multimedia applications. It is particularly relevant in case of a cloud-based solution where specific components in the cloud are responsible for

the rendering of application interfaces and for processing multimedia content for individual users. In this cases, the system must be designed in a way that it scales even for a large number of parallel users and ensuring the availability of required resources.

NF-REQ6: Interoperability

Interoperability is of particular importance in the multiscreen domain since the application may be distributed on devices from different manufacturers and running different software platforms. The challenge here is to identify the minimal set of interfaces, protocols, and vocabularies that can be standardized in order to ensure the interoperability across devices and platforms. This work will be done in standard organizations and institutions like the World Web Consortium W3C [11].

3.3 Conclusion

In this chapter, the most important use cases in the area of multiscreen multimedia applications were identified, and the functional and non-functional requirements were derived. These requirements are considered in the next chapters for the definition of a multiscreen application model, related concepts and design patterns. Also, some of the requirements will result in the specification of APIs that can be used in multiscreen applications to access functions of the underlying platform. The interworking between the runtime environments on different devices is also considered, and the corresponding network protocols are identified. Furthermore, the media-related requirements serve as input for the design and specification of a multimedia playout system that can address devices with different playback capabilities. Finally, the non-functional requirements listed above will be addressed in the proof-of-concept implementation as well as in the evaluation.

4 Multiscreen Application Model and Concepts

This chapter forms the foundation of this thesis and presents a novel model for the development of multiscreen applications and related concepts. Section 4.1 discusses initial ideas for the conceptual design of multiscreen applications based on the use cases and requirements identified in the previous chapter. Section 4.2 introduces a new method called *Multiscreen Model Tree* for modeling multiscreen applications during each stage of the application lifecycle. Section 4.3 presents the different concepts and approaches for interworking between multiple components in a multiscreen application. Section 4.4 introduces the architecture of a multiscreen application platform and related protocols. Section 4.5 focuses on the usage of Web technologies as a cross-platform solution for developing multiscreen applications. Finally, Section 4.6 provides a proof-of-concept implementation of the architecture, the application model and the runtime environment introduced in this chapter.

4.1 Introduction

The use cases and requirements defined in the previous chapter will serve as input for the definition of a unified model for multiscreen applications. One of the common characteristics required in all use cases is the high flexibility and adaptability of multiscreen applications. Parts of a multiscreen application can be freely migrated between screens during runtime, and the number of devices involved can increase or decrease dynamically as devices can join or leave at any time. The multiscreen application model should consider these key characteristics during the design and specification phase. A proper approach to address the flexibility and adaptability characteristics is to consider a multiscreen application as a set of loosely-coupled application components that can be easily migrated between devices during runtime. In the remainder of this thesis, the term MSA is used as an abbreviation for a Multiscreen Application and MSC for a Multiscreen Application Component. A MSA consists of a set of MSCs. Each component MSC_i of a multiscreen application has an internal state S_i, a representation of its view V_i, and a runtime function R_i, i.e., $MSC_i = (S_i, V_i, R_i)$. In order to ensure the flexibility and adaptability, some rules have to be considered:

1. Only the runtime function R_i of a multiscreen component MSC_i can change its state S_i and view V_i.
2. Changes to the internal state S_i of a multiscreen component MSC_i can result in changes to its view V_i.
3. Only the runtime function R_i can interact with other application components running on the same or other devices.
4. Creating two instances of the same application component MSC_i and with the same initial state S_i on devices with the same characteristics results in two identical views.

Multiscreen applications often consist of identical functions for different devices and contexts. Implementing these functions multiple times for each device and application context increases the development and maintenance costs and time. Support of reusable components is a critical factor that prompts us to distinguish between two types of application components, atomic and composite. An Atomic Application Component AAC is the smallest indivisible entity in a multiscreen application and can run together with other atomic components on the same device. Each AAC_i has its own state S_i, view V_i, and runtime function R_i and shares the device display with other atomic components assigned to the same device. The combination of all AACs assigned to the same device builds a Composite Application Component CAC where its state is the combination of the states of all containing AACs. The same applies to views and runtime functions of each atomic application component. A CAC can be seen as an entity that coordinates the execution and resources of the containing AACs. For example, it defines how the AACs can share the screen of the assigned device to display the individual views. It can also act as a broker between the containing AACs assigned to the same device or other CACs running on other devices within the same multiscreen application. Furthermore, a CAC provides functions to add, remove or migrate atomic components.

To illustrate the concept of AACs and CACs, let us consider the *Multiscreen Game* use case from Section 3.1.2 as an example. We can identify from this application scenario two atomic application components, AAC_p representing the player game field (index p in AAC_p is the abbreviation of *player*) and AAC_t representing the common game field or table (index t in AAC_t is the abbreviation of *table*). There are different compositions, how the atomic components are distributed on the screens, depending on the current state of the game and the number of connected players. At the beginning of the game, there is one instance of each AAC and both run on the device of the first player. This means that the multiscreen application has after the first launch ($MSA(t = 1)$ in Figure 4.1) only one composite component instance CAC_{pt1} which consists of two atomic component instances AAC_{p1} and AAC_t as shown in Figure 4.1. It is worth to mention that the notation $MSA(t)$ defines the multiscreen application at time t which increases stepwise after an application component is

added, removed or migrated using one of the operations we will introduce later in Section 4.2. Figure 4.1 shows the multiscreen application at the different time steps. This is just an informal visualization of the application components in order to make the idea of AACs and CACs more understandable. Section 4.2 will provide a better comprehensive approach to visualize the composition of the application components at any stage of the application lifecycle. Back to the multiplayer game example, the AAC_{t1} instance gets migrated in the next step to the TV and runs inside of the newly launched composite instance CAC_t ($MSA(t = 2)$ in Figure 4.1). After the second player joins the game, a new composite instance CAC_{p2} with a single atomic instance AAC_{p2} will be launched on his device ($MSA(t = 3)$ in Figure 4.1). In the next step, the third player joins the game after he received an invitation from the first or second player. Since he is not at the same physical location as the first two players, he needs in addition to a new atomic player instance AAC_{p2} also a new atomic table instance AAC'_t which needs to stay in sync with the first AAC_t instance. $MSA(t = 4)$ in Figure 4.1 shows the state of the multiscreen application after the third player joins the game. Finally, $MSA(t = 5)$ in Figure 4.1 shows the state of the application after the third player migrates the table component AAC'_t from his device to the TV and creating a new composite component instance CAC'_t.

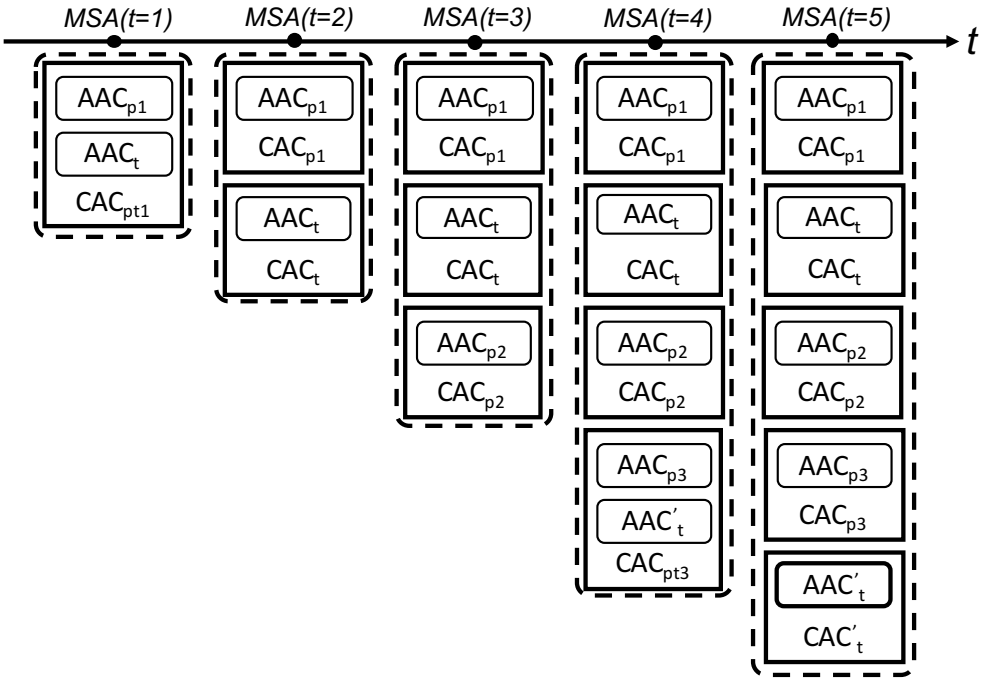


Figure 4.1.: Components of the Multiscreen Multiplayer Game at different Stages

Thus, the basic idea for designing a multiscreen application in the first step is to identify all relevant atomic application components. This can be done by analyzing the application scenario and deriving the requirements from it. The next step is to identify the possible combinations of atomic application components and to find out which composite application components are relevant. If during the design phase

two or more atomic application components always appear together in composite application components, they should be replaced with a new atomic component that provides the same functionalities. It does not make sense to consider these atomic components individually if they always belong together. Each unnecessary atomic component is an additional effort during the conception and later during the implementation of the multiscreen application. An important aspect while identifying the atomic components is to consider the role and functionality of each component rather than how it could be displayed on each device. For example, in the multiscreen game use case, the table component is identified as atomic application component AAC_t and can be displayed on the player device together with the player atomic component AAC_p as part of the composite component CAC_{pt} or as a standalone component on the TV as part of composite component CAC_t. The AAC_t may look different on each device class, but it is still the same logical component. In this case, the component needs to dynamically adapt to the capabilities of the target display like screen resolution and supported input methods. In the next sections, we will go deeper into all multiscreen aspects and address each of the identified requirements. We will first introduce a new method for modeling multiscreen applications called "Multiscreen Model Tree" (MMT), which describes the composition of the components of a multiscreen application, the dependencies between them and their assignment to individual devices. It is worth to mention that the multiscreen model tree is not a method for describing or tracking the state of the entire application or of individual components. There are well-studied concepts and methods such as "State Machines" [123] and "Timed Automata" [124] that can be used for this purpose. For example, [125] uses the concept of timed automata as "*a formal model for the representation of Web Service workflows*".

4.2 Multiscreen Model Tree

As mentioned in the introduction, the multiscreen model tree provides a way to track the components of a multiscreen application and describes the dependencies between them as well as the assignment to individual devices during the application lifecycle. Figure 4.2 shows an example of the multiscreen model tree. The root element of the tree is always the multiscreen application MSA itself. The second level of the tree contains all devices D_i involved in the application. For example, there are four devices D_1, D_2, D_3 and D_4 involved in the multiscreen model tree depicted in Figure 4.2. The third level of the tree contains the composite application components CAC_i assigned to devices D_i from the second level. It is important to know that a device D_i can be either empty or runs only one composite application component CAC_i. A device like D_4 in the example tree is empty which means that it is available, but currently, there is no CAC launched on it. In the example

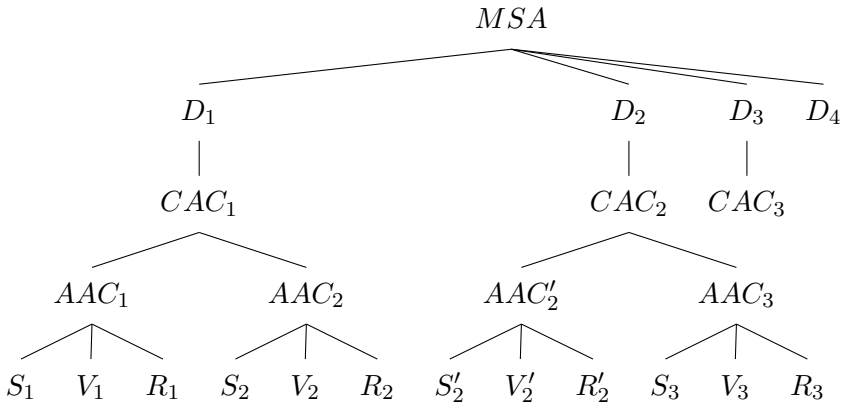


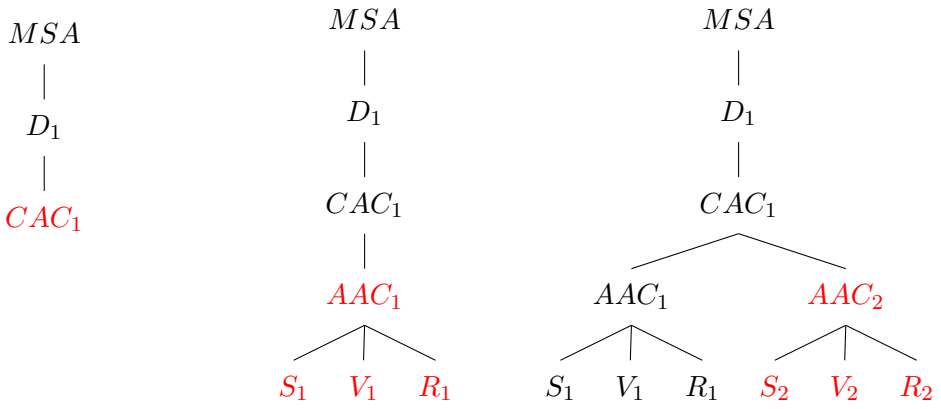
Figure 4.2.: Multiscreen Model Tree Example

model tree, there are three composite application components CAC_1, CAC_2 and CAC_3, each assigned to the devices D_1, D_2, and D_3. The fourth level of the tree contains the atomic application components AAC_j that are part of the corresponding composite application components CAC_i from the third level. An atomic application component instance can be only part of one composite application component instance at a time. A composite application component can be empty like CAC_3 in the example tree which means that the application is assigned to the corresponding device and is ready to add atomic application components to it, but no components are currently added to it. In the example tree, there are two atomic application component instances AAC_1 and AAC_2 as part of CAC_1 and two atomic application component instances AAC'_2 and AAC_3 assigned to CAC_2. In this example, the instance AAC'_2 is a mirror of AAC_2. The notations $'$, $''$, and $'''$ mean that atomic application component instances AAC'_x, AAC''_x, and AAC'''_x are mirrored from the origin instance AAC_x. The following subsections explain how to use the multiscreen application tree to describe various functions of a multiscreen application.

4.2.1 Instantiation

Instantiation is the process of creating and initializing a new multiscreen application or a new multiscreen application component. In case of atomic application components, there are two methods for creating a new instance AAC_x of an atomic application component AAC: 1) the new instance is created and initialized using default data according to the runtime function of the atomic component, or 2) the new atomic application component instance $AAC_x = (S_x, V_x, R_x)$ is created from the current state $S_y(T_1)$ at time T_1 of another atomic application component instance $AAC_y = (S_y, V_y, R_y)$ as initial state of the newly created atomic application component. In other words, the expression $S_x(T_1) = S_y(T_1)$ is correct at time $T_2 = T_1$ but not necessary at time $T_2 > T_1$.
In the case of composite application components, a newly created instance is always

empty. Once a new CAC instance is created, then atomic application components can be added to it. In other words, to create a new CAC_x instance from an existing instance CAC_y which already contains atomic component instances, it is necessary to create a new AAC_y instance for each AAC_x instance and add it to CAC_y. The



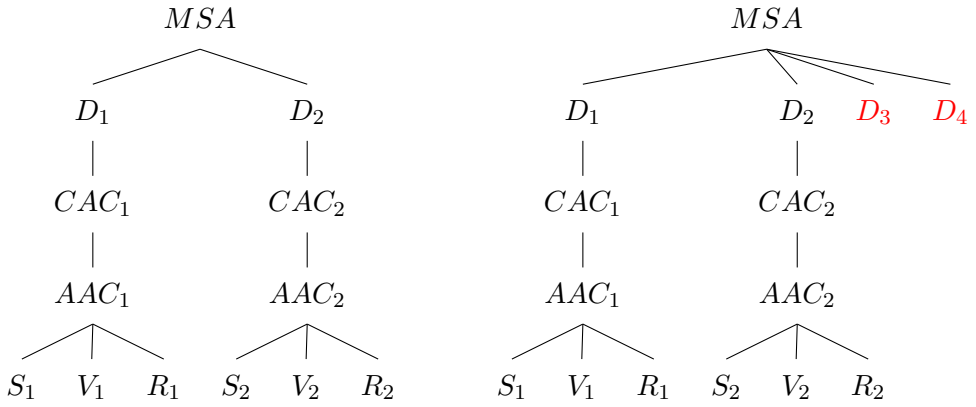
(a) MSA Model Tree after Instantiation of CAC_1 **(b)** MSA Model Tree after Instantiation of AAC_1 **(c)** MSA Model Tree after Instantiation of AAC_2

Figure 4.3.: Multiscreen Model Tree: CAC and AAC Instantiation

initial state of a multiscreen application is always a single screen application which contains a single composite component assigned to the device on which the user started the application. Figure 4.3a shows a minimal multiscreen application tree immediately after instantiating a new CAC. This may trigger the instantiation of atomic component instances as depicted in Figures 4.3b and 4.3c where the two instances AAC_1 and AAC_2 are created and added to CAC_1. During runtime, the single screen application can turn itself in a multiscreen application after discovering new devices and launching new composite component instances on them. New atomic application instances can be instantiated and added to newly launched composite instances as well. All these steps will be discussed and described in the next sections.

4.2.2 Discovery

A multiscreen application is intended to run on multiple devices simultaneously. As mentioned in Section 4.2.1, a multiscreen application runs on a single device after the first launch similar to any single screen application. During runtime, the single screen application can discover other devices and launch other application components on them. Thereby, devices may appear and disappear at any time depending on many factors like availability and reachability. The multiscreen application should take these factors into account and expect changes in device availability during application lifecycle. Therefore, a multiscreen application should be able to discover other devices during runtime either on demand or via notification when suitable



(a) Before discovery **(b)** After discovery

Figure 4.4.: Multiscreen Model Tree before and after discovery

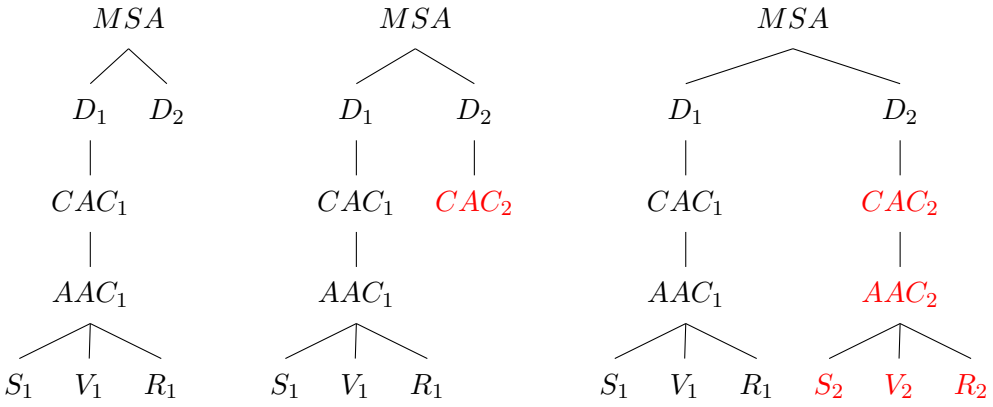
devices become available. The multiscreen application should also get notified when an already discovered device disappears. The application can keep its internal list of discovered devices in sync with physically existing devices and avoid unexpected behavior in case devices are not available but still exist in the list of discovered devices.

Discovered devices can be tracked in the multiscreen model tree by adding a new node for each discovered device to the second level of the tree. The nodes of newly discovered devices do not have child nodes as shown in Figure 4.4b. Figure 4.4a shows the multiscreen model tree before triggering the discovery process. The children of device nodes are always composite application instances that can only be added after the launch step. One important aspect of the discovery is the capability to find only devices that fulfill specific requirements. This facilitates avoiding runtime errors in case a discovered device does not support a specific mandatory feature in order the application works properly. For example, in the remote media playback use case defined in Section 3.1.1, the component running on the mobile device may request to discover only devices that support specific video and audio codecs.

The discovery can be triggered by any atomic application component already assigned to a device. The discovery request will be forwarded to the parent composite application component which calls the underlying discovery API on the assigned device. Once the list of discovered devices is available, the multiscreen application provides the result to the component that triggered the request and optionally to other components of the application. Composite application components can now be launched on any of the newly discovered devices. The launch process is described in the next section.

4.2.3 Launching and Terminating of Application Components

After a device is discovered, the application component that initiated the discovery request will get all information necessary for launching application components on it. Which information is required depends on the underlying technologies. In most cases, a "friendly name" of the discovered device will be displayed to the user to distinguish discovered devices from each other when multiple devices are available. Based on the discovery information, the requesting application component AAC_1



(a) Before launch **(b)** After launch of CAC_2 **(c)** After launch of AAC_2

Figure 4.5.: Multiscreen Model Tree before and after launch

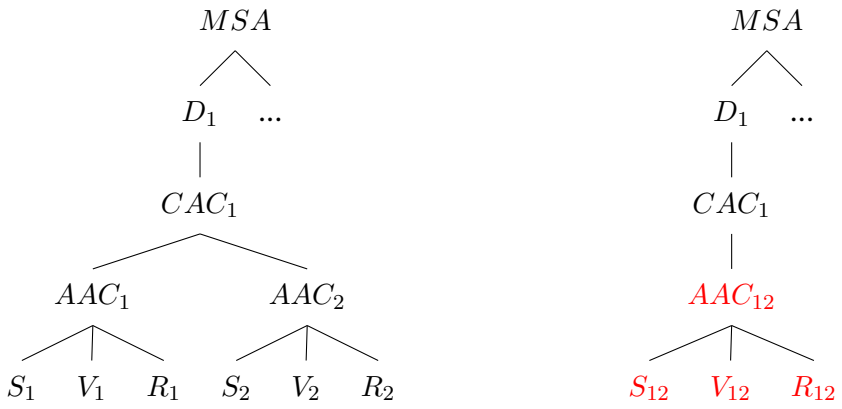
can now initiate a request for a specific composite application component CAC_2 to be launched on a selected device D_2 as shown in Figures 4.5a and 4.5b. Once CAC_2 is launched, the requesting application component AAC_1 and optionally other components of the same multiscreen application will get notified and the atomic application component AAC_2 can be added to the newly launched composite component CAC_2 as depicted in Figure 4.5c.

After the launch is completed, the requesting atomic application component AAC_1 and the newly launched atomic application component AAC_2 will be able to interact with each other using different methods like establishing an application-to-application (App2App) communication channel between the two components, using a publish/subscribe paradigm or by following a data-centric approach where the state of the multiscreen application is synchronized between all devices on which the application is running. All these approaches will be discussed in following sections of this chapter.

Similar to the launch feature, any application component can terminate other application components running on remote devices. In this case, all connections established to the terminated component will be closed, and affected components will be notified.

4.2.4 Merging and Splitting

Atomic application components are the smallest entities in a multiscreen application. Their main purpose is to build applications from modular components that can be freely moved between devices and even be reused in different applications. This means that multiple atomic application components may run inside the same composite application component on the same device. Let us consider the two atomic components $AAC_1 = (S_1, V_1, R_1)$ and $AAC_2 = (S_2, V_2, R_2)$ which both run inside the composite component CAC_1 as depicted in Figure 4.6a. There are two



(a) Before merging **(b)** After merging

Figure 4.6.: Multiscreen Model Tree before and after merging

options for running the atomic components AAC_1 and AAC_2 on the same device:
Option 1: The atomic application components AAC_1 and AAC_2 run inside of CAC_1 independent of each other in two different and isolated execution contexts. AAC_1 and AAC_2 can interact with each other in the same way as if they were running on two different devices. Since both components AAC_1 and AAC_2 are sharing the same screen, CAC_1 needs to coordinate how the views V_1 and V_2 are rendered on the device's display. This simplest approach to do this is to assign parts of the screen as rendering areas for each view. Another approach is to assign the whole screen to each view but as a different layer. All layers have a transparent background and are placed on top of each other. The application developer can define the logic for the layering in CAC_1. An atomic component may be notified after adding and removing other atomic components to the same composite component which allows adapting the views to the new context.
Option 2: The atomic components AAC_1 and AAC_2 are replaced with a new atomic component AAC_{12} which provides the same functionality as if AAC_1 and AAC_2 were running simultaneously on the same device. In other words, the components AAC_1 and AAC_2 are merged into a new atomic component AAC_{12} as depicted in Figure 4.6b. This is needed if the first option is not applicable, for example, if the application developer needs to customize the application UI in a very flexible way

where each AAC can control any part of the screen and not only a pre-defined area. In this case, a new view $V_{12} = V_1 + V_2$ is created by merging V_1 and V_2. We will use the $+$ operator for the merge operation. The states S_1 and S_2 as well as the runtime functions R_1 and R_2 can still run simultaneously in two different execution contexts as described in the first option. This means that the new runtime function is $R_{12} = R_1|R_2$ and the new state is $S_{12} = S_1|S_2$ where $|$ designates parallel execution. Finally $AAC_{12} = (S_1|S_2, V_1 + V_2, R_1|R_2)$. Any changes in the states S_1 or S_2 may lead to changes in V_{12}.
The second option describes the most important combination for merging two AACs by merging their views and keeping the states and runtime functions running in different contexts. This is important since the application may request to split the merged AACs again at any time later, for example, to migrate one of the AACs to another device. In this case, it is easier only to split the views instead of splitting the states and runtime functions if they were merged before. However, in extreme situations, for example, when a merged component performs better and more efficient than when each AAC is running in a separate context, it is possible to replace the source AACs with a completely new $AAC_{12} = (S_1+S_2, V_1+V_2, R_1+R_2)$. This should be avoided if possible since in this case the developer needs to implement a new component AAC_{12} in addition to the source components AAC_1 and AAC_2.

As stated before, splitting is the inverse operation of merging which allows an existing AAC to be divided into two AACs that can be executed separately in two different runtime contexts. In most cases, this is needed when part of an AAC needs to be migrated to another device. For example, in the multiscreen game use case described in Section 3.1.2 after the first player launches the game, both atomic components AAC_p (player component) and AAC_t (table component) will run together on the user device as merged AAC_{pt} where both views V_p (player view) and V_t (table view) share the same screen. When the player decides to migrate the table component to the TV, the AAC_{pt} first needs to be split into AAC_p and AAC_t where AAC_p stays on the player device and the AAC_t is migrated to the TV. The concept migration will be described in the next section.

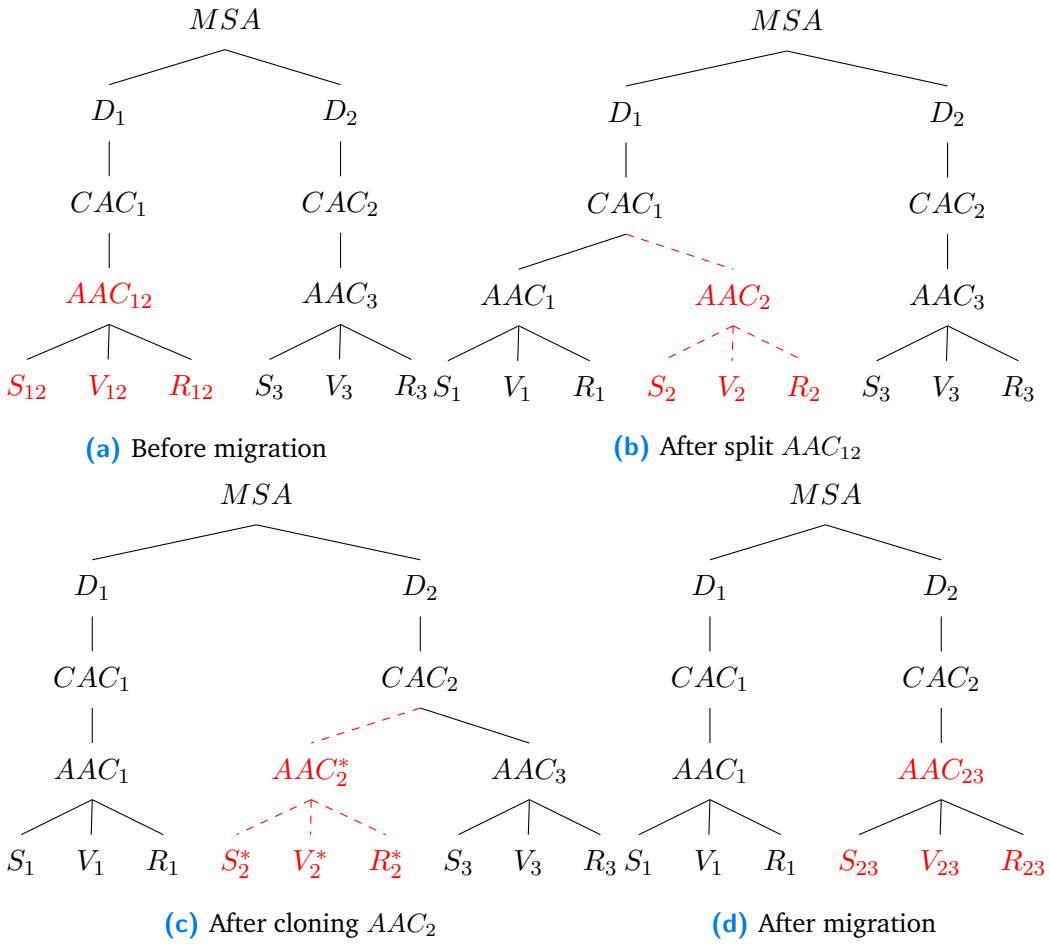
4.2.5 Migration

Migration is defined the process of moving an atomic application component from one composite application component CAC_1 running on a device D_1 to another composite application component CAC_2 running on a device D_2. Migration can be completed in four steps where the first and last steps are optional.

1. If the atomic component under consideration AAC_2 is part of a merged component AAC_{12} as depicted in Figure 4.7a, then the split operation described in

the previous section needs to be applied. Therefore, AAC_{12} will be replaced by the two atomic components AAC_1 and AAC_2 as depicted in Figure 4.7b.

2. In next step, the atomic component AAC_2 will be detached from CAC_1. This means that the state S_2 of AAC_2 will remain available, but the view V_2 will no longer be displayed as shown in Figure 4.7b (dotted line). Furthermore, the runtime function R_2 will be suspended, i.e., no changes will be made to the state S_2 anymore.
3. Launch a new instance AAC_2^* on CAC_2 assigned to device D_2 using the state S_2 as the initial state. The view V_2^* will be attached to CAC_2, and the runtime function R_2 will be resumed which means that it can make changes to the application state S_2^*. At the same time, the atomic component AAC_2 will be removed from CAC_1.
4. Merge the AAC_2^* with existing atomic components running on CAC_2 if necessary.



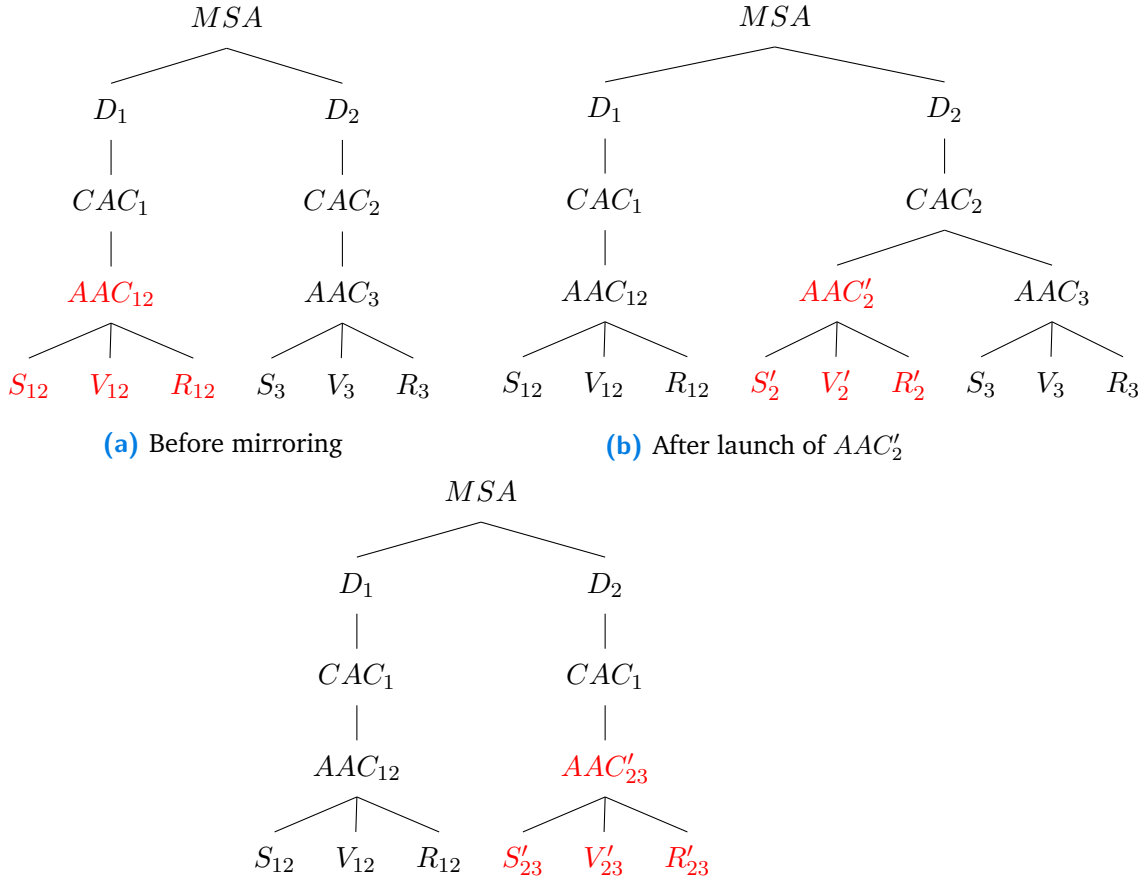
(a) Before migration **(b)** After split AAC_{12}

(c) After cloning AAC_2 **(d)** After migration

Figure 4.7.: Multiscreen Model Tree before and after Migration

4.2.6 Mirroring

Mirroring is the process of cloning an atomic application component instance AAC_1 assigned to a composite application component CAC_1 running on device D_1 and launching the cloned atomic instance AAC_1' on another composite component CAC_2 running on device D_2. At any time after the cloning, both components CAC_1 and CAC_1' must keep their states synchronized, which will also imply their views. Similar to migration, the mirroring can be completed in three steps, where the first and last steps are optional:



(a) Before mirroring

(b) After launch of AAC_2'

(c) After mirroring

Figure 4.8.: Multiscreen Model Tree before and after mirroring

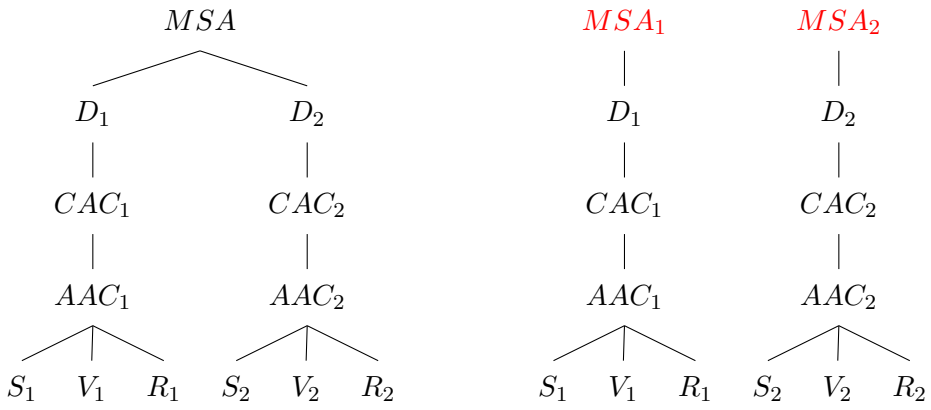
1. If the atomic component AAC_2 under consideration is part of a merged component AAC_{12} with a merged state S_{12} as shown in Figure 4.8a, then the state S_2 of AAC_2 needs to be determined after splitting the state S_{12} without making any changes on AAC_2 .
2. Launch a new instance AAC_2' on CAC_2 assigned to device D_2 using the state S_2 as the initial state. Furthermore, the view V_2' will be attached to CAC_2,

and the runtime function R_2 will be resumed which means that it can make changes on the application state S_2'.

3. Merge the AAC_2' with existing atomic components running on CAC_2 if necessary. This step is not necessary in cases there is no need to merge components.

4.2.7 Joining and Disconnecting

Disconnecting is the step when a composite application component CAC_1 running on device D_1 closes its connections to all other composite application components running on other devices as shown in Figure 4.9. Therefore, the multiscreen application will be split into two parts that run separately from each other. The disconnection may occur either on demand upon user request or due to an unexpected problem. The most relevant example for disconnecting is the remote playback use case. In most situations, the user uses the smartphone to search for media content and the TV to playback selected media. The user can also disconnect the smartphone from the TV without stopping the playback and connect again at any time later. Joining



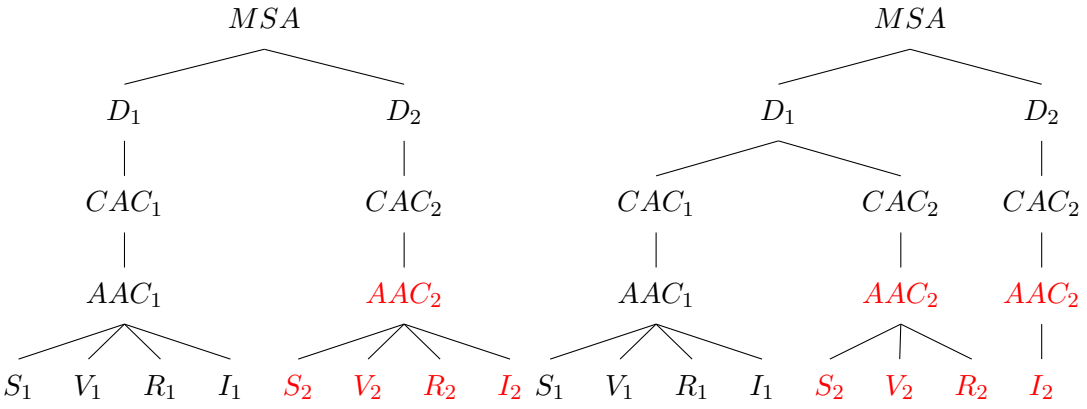
(a) Before disconnecting **(b)** After disconnecting

Figure 4.9.: Multiscreen Model Tree before and after disconnecting

is the opposite operation of disconnecting. It allows an application running on one device to connect to another application that runs on a second device. The result is a multiscreen application containing all components of both source applications. Joining may occur after the disconnecting step, for example, in the remote playback use case described above, the disconnected control application may connect again to the player application running on the TV. However, there are cases where joining does not occur necessarily after disconnecting. For example, a companion screen application can connect to a hybrid broadcast application launched on the TV automatically after the user switches to the corresponding channel.

4.2.8 Rendering

In the previous sections, we considered an atomic application component as a triple (S, V, R) that consists of a state S, a view V and a runtime function R. We considered implicitly that the view V is rendered on the same device to which the parent composite application component is assigned. Rendering is the function for creating the image output I of the application interface at a specific time. There are new emerging technologies that allow to render the application output on one device or in the cloud and to send the image output and display it on a second device. This is relevant on low-capability devices that are not capable of rendering the application interface by themselves. During the modeling and design phase



(a) Local Rendering **(b)** Remote Rendering

Figure 4.10.: Local and Remote Rendering

of a multiscreen application, it is not important to know where the application runs and where it is displayed, but this is relevant when it comes to identifying the right architecture based on given non-functional requirements. Therefore, we will extend the Multiscreen Model Tree to distinguish between the different rendering options. A new optional element I which represents the rendering function at a given time will be added to an atomic application component $AAC = (S, V, R, I)$. This means that the same AAC can be part of two different composite application components. For example, Figure 4.10a shows a multiscreen application where each device renders the UI of CAC_1 and CAC_2 locally while Figure 4.10b shows a multiscreen application where device D_1 renders the UI of CAC_1 and CAC_2 locally, but only the image output of CAC_2 is displayed on device D_2.

4.3 Multiscreen Application Concepts and Approaches

In previous section, we discussed and introduced a new method for modeling a multiscreen application using a tree-based structure. This model allows us to capture the state of a multiscreen application at every stage of its lifecycle, regardless of the underlying platform and development paradigm that are presented in this and next sections.

4.3.1 Message-Driven Approach

As the name of this approach suggests, the main idea is to enable collaboration and interaction between atomic application components running on the same or different devices by exchanging messages between the components. In order to establish a communication channel between two atomic application components AAC_1 (the sender) and AAC_2 (the receiver), AAC_1 must know the receiver component AAC_2 with all related information like the end-point to open the communication channel. In this section, we will discuss the concepts and approaches apart from the technical details. There are three options for how the sender can get the required information:

1. **After launch**: If the atomic component AAC_1 was the component that triggered the launch of AAC_2, then it should receive all information needed to establish a communication channel to the newly launched component. For example, in the multiscreen gaming use case described in Section 3.1.2, the application running on the device of the first player launches the table component on the TV and can immediately establish a communication channel to it.
2. **After discovery**: if the atomic component AAC_1 wants to connect to an already launched atomic component instance from a specific type, then it should trigger a discovery request using the component type as a filter. If multiple devices are available, the user will be requested to select one of them. For example, in the same multiscreen gaming use case, the application running on the device of the second player discovers the application running on the TV and gets the necessary information about the table component to establish the communication channel to it.
3. **After invitation**: If the atomic component AAC_1 was launched after receiving an invitation from a component already in the multiscreen application, then AAC_1 can use the information sent in the invitation to connect to the desired receiver component AAC_2. For example, also in the same multiscreen gaming

use case, the components running on first or second player devices that are already in the game can send an invitation (using an out-of-band communication channel) to a third player to join the same game. The third player launches the player component which uses the information from the invitation to join the same game.

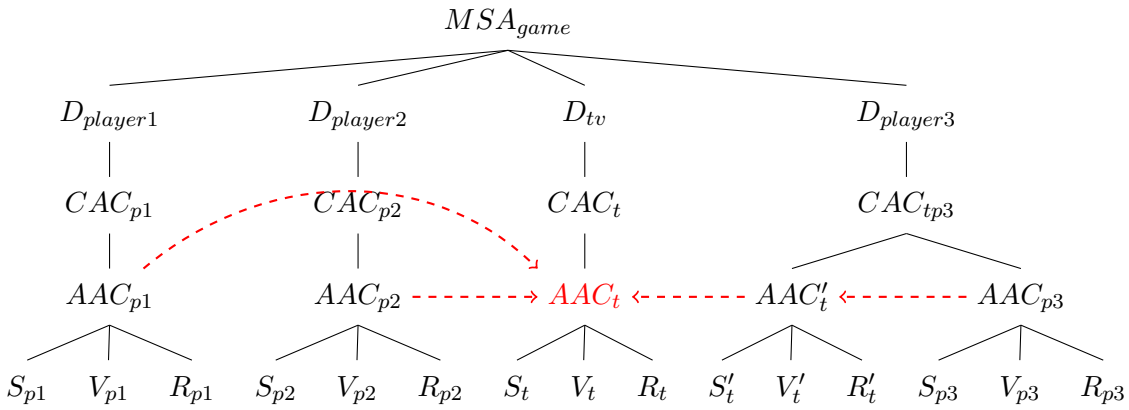


Figure 4.11.: Multiscreen Model Tree of a Multiplayer Game following the Message-Driven Approach

Designing a multiscreen application using the message-driven approach should be aligned to the following rules:

1. Identify atomic application component classes. In the multiscreen model tree example depicted in Figure 4.11 and the diagram depicted in Figure 4.12, we can identify the two atomic component classes AAC_p and AAC_t for player and table components.
2. Identify the number of instances that can be created for each atomic component. In the game example, a new AAC_p player instance will be created for each user. Furthermore, at least one AAC_t table instance should be created, and all table instances should stay in sync.
3. Identify the atomic component instances that could play a master role. The master is capable of coordinating the interworking between the components. In the gaming example, the first created AAC_t table instance is a good candidate to play the master role.
4. Identify the sender and receiver components. In most situations, the master component can be the receiver and the other components the senders. In the gaming example, the AAC_t component takes the receiver role, and all other components are senders.
5. Identify the data that should be kept in the state of each atomic component. Some data can be stored redundantly across multiple components. In the gaming example, the game state can be stored on AAC_t while the AAC_p stores the state of each player component.

6. Identify the messages and commands that can be exchanged between the atomic components. In the gaming example, the component of the player who is currently playing sends a message to the table component containing the performed action with other related information such as the cards played. The table component sends a notification message to all other components in order to update their internal states and to select the next player.

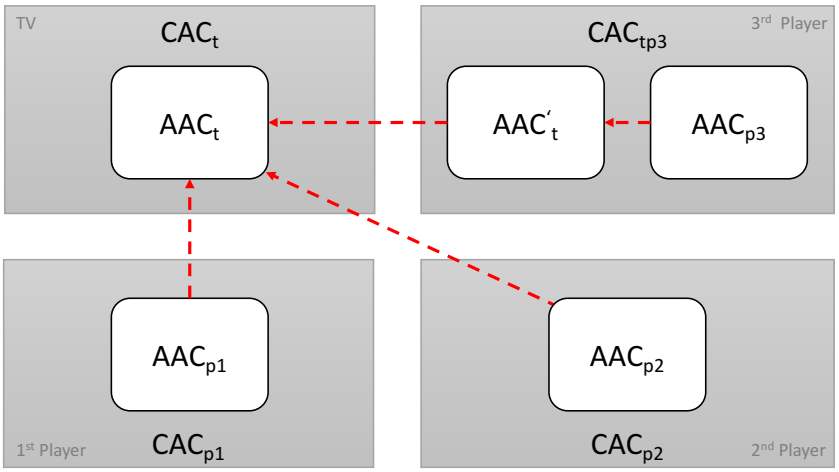


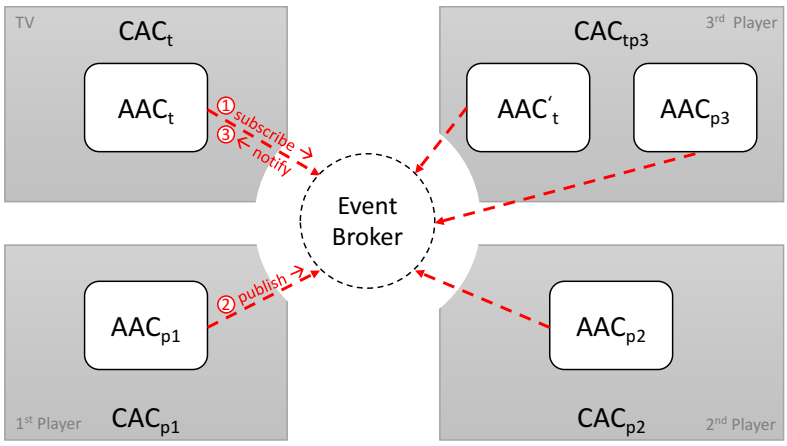
Figure 4.12.: Message-Driven Approach

In summary, in the message-driven approach, the multiscreen application is responsible for keeping the states of all atomic application components in sync by exchanging messages through the established communication channels. This also means that the multiscreen application is responsible for keeping the states of mirrored instances of an atomic component in sync with the origin atomic component. Furthermore, migration of atomic application components between devices will be handled entirely on the application level. If an atomic component gets migrated from one device to another, then each atomic component which was already connected to the component before the migration should reconnect to the new atomic component instance after migration.

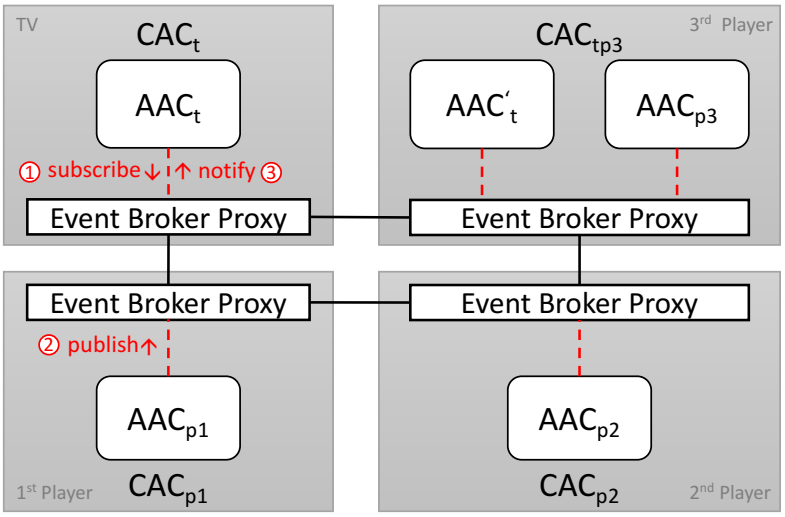
4.3.2 Event-Driven Approach

The event-driven approach for developing multiscreen applications addresses the challenges of the message-driven approach especially for establishing and maintaining communication channels between atomic component pairs on the application level. It can become more complicated if the atomic components often migrate between devices and the affected components need to reconnect to the migrated components. The main idea of the event-driven approach is to follow another concept that does not require a logical communication channel between two atomic components. In contrast to the message-driven approach, the event-driven approach

allows any atomic component to access an "event broker" entity which offers two main functions: one for subscribing to events of specific types and another one for publishing events of specific types. An event is defined as $E = (T, D, P)$ where T is the type of the event, D is the data or content of the event, and P is the publisher of the event which is the identifier of the atomic component instance that published the event. Designing a multiscreen application using the event-driven approach is



(a) Centralized Event Broker



(b) Decentralized Event Broker

Figure 4.13.: Event-Driven Approach

similar to the message-driven approach defined in Section 4.3.1 except the following differences:

- An atomic component can interact with other atomic components running on the same or different devices without knowing the end-points of these components or the need to handle the reconnection in case a component migrates from one device to another. Instead, an atomic component only needs to subscribe to event types of interest or publish events using the event broker.

- The developer needs to identify all relevant event types, the structure and format of the data published with each event instead of identifying the messages that can be exchanged between two atomic components when using the message-driven approach.

As depicted in Figure 4.13, we can see that there are two different architectures for realizing the event driven approach and both offer the same $publish/subscribe$ operations for the atomic application components. This means that the selected architecture will not affect the conceptual design and development of the multiscreen application, but will only have an impact on the underlying implementation of the event broker and related $publish/subscribe$ operations. The two architectures are:

- **Centralized Event Broker**: The centralized event broker architecture is depicted in Figure 4.13a. There is one central entity that plays the role of the event broker and this entity is well known to the underlying runtime on each device. The event broker may run on a central server in the cloud, on a dedicated server in the local network or on a dedicated master device of the multiscreen application.
- **Decentralized Event Broker**: The decentralized event broker architecture is depicted in Figure 4.13b. There is no central event broker, but instead, each device involved in the multiscreen application runs an event broker proxy. All event broker proxies are connected with each other and build a virtual event broker. An event broker proxy offers the same $publish/subscribe$ operations as the event broker in the centralized architecture.

The event-driven approach makes the conceptual design and development of multiscreen applications simpler compared to the message-driven approach since it hides the complexity of using dedicated communication channels between any two atomic application components. On the other hand, the underlying implementation of the event-driven approach on the platform level is more complicated than the message-driven approach especially if the decentralized approach is selected. These challenges will be discussed in more details in the implementation section.

4.3.3 Data-Driven Approach

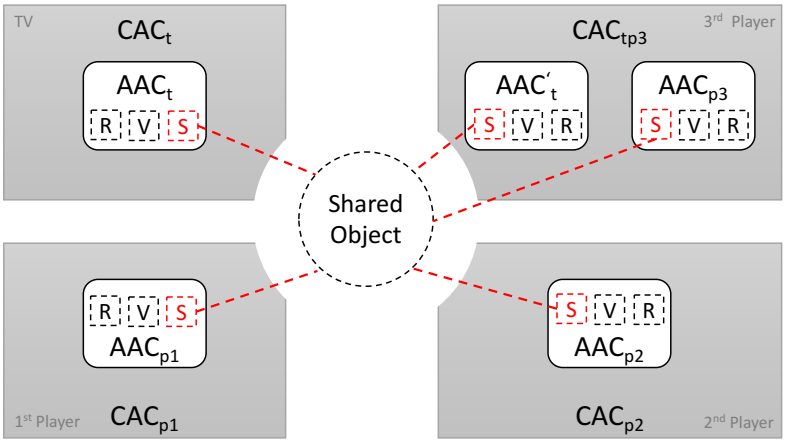
The data-driven approach addresses the synchronization challenges which are not solved in the message-driven and event-driven approaches. The synchronization of the application state across multiple atomic components can be implemented on top of the messaging channels in case of message-driven approach or on top of events in case of event-driven approach. The data-driven approach addresses this aspect and integrates the synchronization functionality on the platform level instead of

letting application developers deal with it on the application level. The basic idea of this approach is to let the runtime function R of an atomic component operate only on the state object S of the same component without the need to interact with other components via dedicated events or messages. The underlying platform will synchronize the state object or part of it of one atomic component with the state objects of other atomic components in the same multiscreen application. Other than in the event-driven or message-driven approaches where the state S of an atomic component is entirely under control of the runtime function R, in the data-driven approach the atomic component should expect that the underlying platform can also manipulate the state S. Designing a multiscreen application using the data-driven approach has the same rules as the event-driven approach defined in Section 4.3.2 (which also includes the rules of the message-driven approach defined in Section 4.3.1) except for the following:

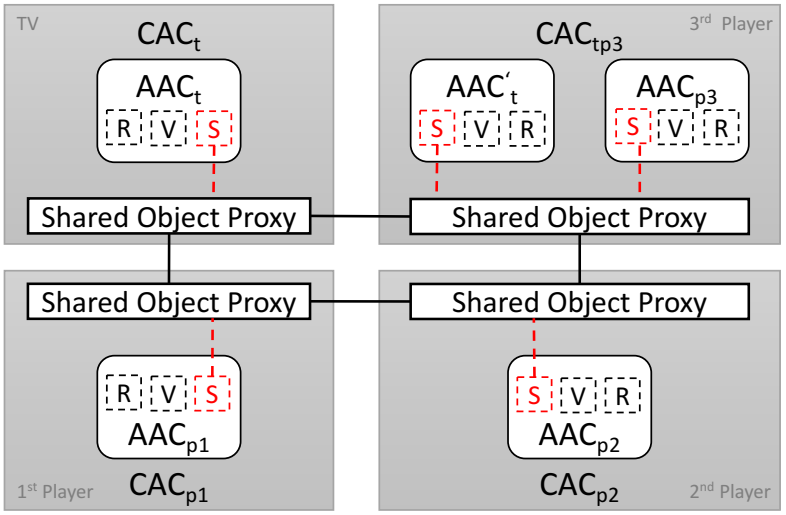
- There is no need for atomic components to interact via messages or events with each other. The atomic component only needs to operate on its state S which will be synchronized automatically with the corresponding elements of the shared object.
- The developer needs to identify the data structure of the shared object and which atomic component can read or write which elements of the shared object.

The data-driven approach introduces a new operation that allows the runtime function R of an atomic component to observe changes in the state object S or any of its properties. All state objects or sub-objects that are subject to synchronization comprise the so-called shared object that holds the state of the whole multiscreen application. As in the event-driven approach, there are also two architectures for the data-driven approach as depicted in Figure 4.14:

- **Centralized Shared Object**: The centralized shared object architecture is depicted in Figure 4.14a. There is a central entity that holds the shared object and keeps local state objects in sync with it. Each manipulation on a local state object will be first applied on the centralized shared object before the changes are applied on the local state objects, and registered observers are notified. Similar to the event broker, the centralized shared object may run on a central server in the cloud, on a dedicated server in the local network or on a master device involved in the multiscreen application.
- **Decentralized Shared Object**: The decentralized shared object architecture is depicted in Figure 4.14b. There is no centralized shared object, but instead, each device involved in the multiscreen application runs a shared object proxy. The shared object proxies are connected with each other, and any change to a



(a) Centralized Shared Object



(b) Decentralized Shared Object

Figure 4.14.: Data-Driven Approach

local state object will propagate in the network until all affected state objects are updated, and all observers are notified.

In both approaches, conflicting changes and state inconsistencies may occur since the object can be manipulated from various clients simultaneously. There are already well-known synchronization algorithms that address these issues:

- **Lockstep Synchronization**: The lockstep synchronization [126] follows a pessimistic approach for synchronizing the state of a shared object in centralized or decentralized systems. The state of the shared object advances step-wise. This means that each client needs to issue in each step an event to the entity managing the shared object and not proceed until an acknowledgement event is received from all other clients. The acknowledgement event includes also the changes made to the object in last step so that each client or peer can update its local copy of the object. Concurrent changes in the same step are resolved or

rejected by the managing entity and can be applied in a sequence or in parallel using transactional memory approaches [127] [128]. A disadvantage of the lockstep synchronization mechanism is that it depends on the performance of the client with the highest network latency or lowest processing capability.

- **Bucket Synchronization**: The bucket synchronization algorithm [129] is an improvement of lock synchronization by allowing clients to not wait for acknowledgement events before they can proceed. The timeline is divided into time buckets of fixed length based on the client or peer with the highest latency. The timelines on all clients are synchronized with a global clock using the Network Time Protocol (NTP) [130]. The bucket synchronization algorithm follows an approach for delaying events for a time that is long enough to avoid incorrect ordering before execution. Inconsistencies can still occur if events are lost or arrive late.
- **Time Warp Synchronization**: Time warp synchronization [131] follows an optimistic approach by allowing peers to execute events on their local copies of the object while taking a snapshot of the state before each execution. If an earlier event is received, a rollback to the last snapshot before the time of this event will be performed and the events occurred after the snapshot time will be re-executed. Anti-messages are sent during rollback to cancel events which become obsolete. A drawback of this algorithm is the high memory usage to keep snapshots of the state and received events. Also the cancellation of events during the rollback can trigger a rollback on other peers and lead to a high number of anti-messages transmitted over the network.
- **Trailing State Synchronization**: Trailing state synchronization [132] improves the time warp synchronization in terms of memory and processing usage by reducing the number of snapshots taken of the state. Instead of keeping a snapshot after executing each command, the trailing state approach keeps snapshots at different simulation times. These snapshots are called trailing states and are intentionally delayed (with different delay times). All received events are immediately applied to the main state of the application and scheduled to be applied on the trailing states with fixed delays. If an event arrives that causally precedes events waiting for application to a trailing state, then the new event and all waiting events will be immediately applied to the trailing state and it becomes the main state.

In summary, the data-driven approach addresses the synchronization challenges and moves the complexity from the application level to the platform level. This will also make the migration of atomic components between devices more straightforward than in the message-driven or event-driven approaches since the local state of an atomic component will not get lost during migration and can be restored from the shared object on the target device. On the other side, the data-driven approach has its drawbacks depending on the selected synchronization algorithm. This selection

depends on multiple factors like latency, bandwidth, memory usage and performance. For example, the lockstep and bucket synchronization algorithms can be selected in multiscreen application scenarios where all devices are connected to the same network and the latency is expected to be very low or the application scenario can tolerate higher latency. For multiscreen applications like real-time multiplayer games where the state is target to be changed in high frequency, optimistic approaches like trailing state synchronization are the better choice. The application must in this case tolerate inconsistencies in the state. Therefore, the underlying implementation of the data-driven approach should support multiple synchronization algorithms and allow developers to select the synchronization algorithm that best suits their needs.

4.4 Multiscreen Platform Architecture

This section defines the architecture of the multiscreen platform by considering the application model and concepts discussed in the previous sections. The architecture of the platform which runs on any device participating in a multiscreen application is shown in Figure 4.15 and consists of the three layers *Multiscreen Application Runtime*, *Multiscreen Application Framework* and *Multiscreen Network Protocols*. These layers will be discussed in detail in the following subsections.

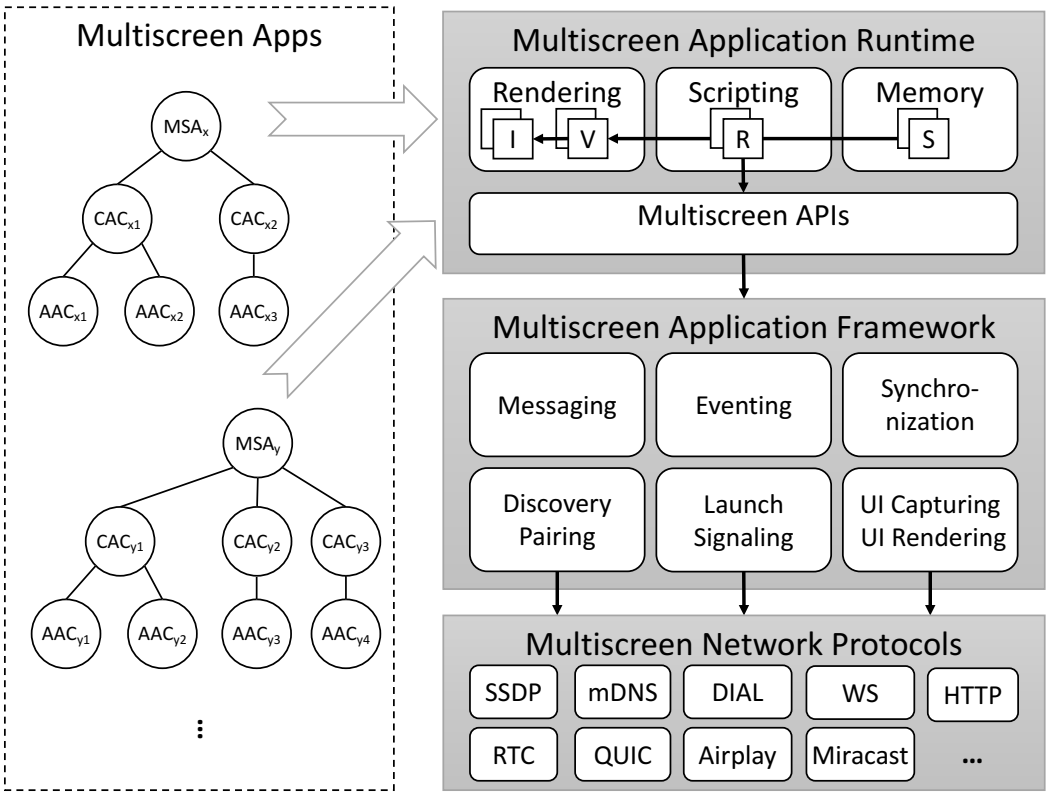


Figure 4.15.: Multiscreen Platform Architecture

4.4.1 Multiscreen Application Runtime

The Multiscreen Application Runtime, or in short the App Runtime, is responsible for executing a CAC and its children AACs on a specific device. Each device involved in a multiscreen application should implement all three layers of the Multiscreen Platform Architecture including the App Runtime.

The App Runtime consists of the four modules *Rendering*, *Scripting*, *Memory* and the *Multiscreen APIs*. The Scripting engine executes the runtime function R of an AAC and holds its state S in a dedicated memory. Furthermore, the App Runtime consists of a Rendering Engine which is responsible for displaying the view V on the device where the atomic component AAC is running. The rendering engine visualizes periodically in a fixed time interval the image output I of the view V on the graphics output interface.

In order to access the multiscreen functions offered by the underlying framework, the App Runtime offers a set of high-level APIs that allow the Runtime function R of an AAC to make use of these functions without the need to deal with the complexity of the underlying system interfaces and protocols. The architecture introduced in this section abstracts from specific technologies used for implementing the applications, the OS running on the target device and the underlying network protocols. In the implementation section, we will discuss the realization of this architecture with a focus on web technologies. In this case, the App Runtime will be just a Web browser extended to the Multiscreen APIs. The AACs are realized as Web applications by using $HTML$ and CSS for implementing the view V, JavaScript for implementing the runtime function R and JSON as the format for recording the state S.

There are different mechanisms for executing and rendering multiscreen application components based on the location where the runtime function R of each component is running and where the corresponding view V is rendered and displayed. The most three important mechanisms *Multiple Execution Contexts*, *Single Execution Context*, and *Cloud Execution*, will be discussed in the following by considering a multiscreen application MSA with two composite components CAC_1 (containing one atomic component AAC_1) and CAC_2 (containing one atomic component AAC_2) and assigned to devices D_1 and D_2.

Multiple Execution Contexts Figure 4.16 shows the App Runtime of the multiscreen application MSA on devices D_1 and D_2. We can see that each device executes, renders and displays its composite application component and the child atomic application components in its own App Runtime within a separate execution context. Both atomic components AAC_1 and AAC_2 can interact with each other using one of the approaches introduced in the previous section via the Multiscreen API, which offers interfaces to the underlying framework layer. Google Cast [9] and DIAL [39] are two state-of-the-art technologies for this mechanism.

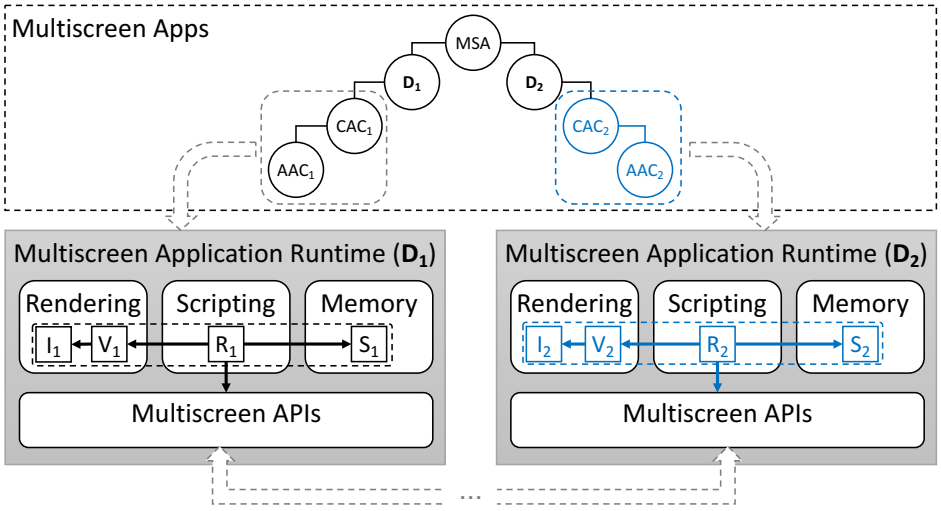


Figure 4.16.: Multiscreen Application Runtime - Multiple Execution Contexts

Single Execution Context Figure 4.17 shows the App Runtime of the multiscreen application MSA on devices D_1 and D_2. As we can see, device D_1 executes,

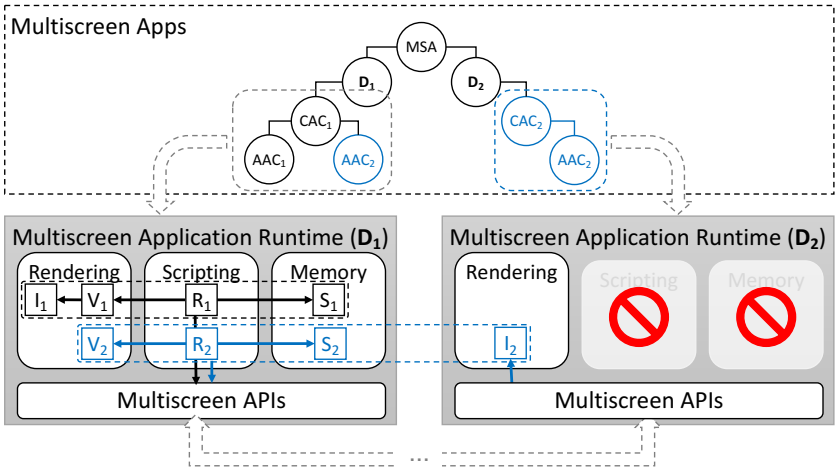


Figure 4.17.: Multiscreen Application Runtime - Single Execution Context

renders, and displays the composite application component CAC_1 and its child atomic application component AAC_1, but only executes and renders the composite application component CAC_2 and its child atomic application component AAC_2. The rendering happens without displaying the UI output on device D_1 which is also called "silent rendering". The rendered image I_2 of CAC_2 will be captured on device D_1 and sent to device D_2 for display. Since the execution of both components happens on a single device, this mechanism is called Single Execution Context. Miracast [8] and Airplay [6] are two state-of-the-art technologies for this approach. Most of these technologies support connecting only to one device at a time.

Cloud Execution Figure 4.18 shows the App Runtime of the multiscreen application MSA on devices D_1 and D_2 as in previous examples. In addition to the previous

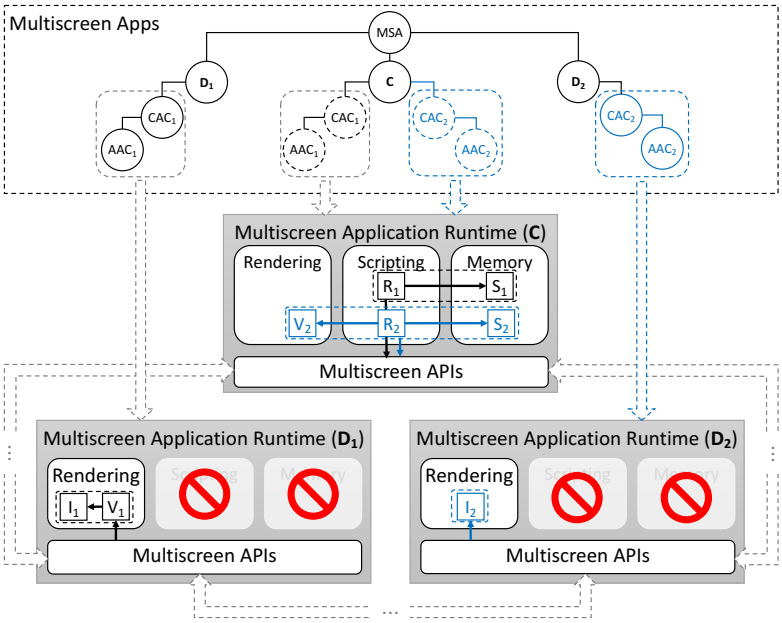


Figure 4.18.: Multiscreen Application Runtime - Cloud Execution

two mechanisms, this method involves a new entity C which runs applications in the cloud in headless mode. In this example, C executes the composite application components CAC_1 and CAC_2 and their child atomic application component AAC_1 and AAC_2. It also renders the view V_2 of AAC_2 and sends the rendered UI I_2 to device D_2 for display while device D_1 renders the view V_1 and displays the output I_1. Cloud Browser [133] and Cloud Gaming platforms like Google Stadia [134] are two technologies that implement this mechanism.

Table 4.1 compares the three App Runtime mechanisms according to various aspects. This is a high-level comparison, and all measurable metrics will be considered in the evaluation section. The color Green in the table represents the best result, Red the worst value and Blue in the middle between both. An explanation for each result in the table is given below:

	Multiple Execution Contexts		Single Execution Context		Cloud Execution	
	D_1	D_2	D_1	D_2	D_1	D_2
Processing	Medium		High	Low	Medium	Low
Software Maintenance	High		High	Low	Low	Low
Disconnection Allowed	Yes		No		Yes	
Multiple Connections	Yes		No		Yes	
Scalability	High		High		Medium	Low
Battery Lifetime	Medium		Low	Medium	Medium	
Motion-To-Photon Latency	Low		Medium		High	
Offline Capability	Yes		Yes		No	

Table 4.1.: Comparison of the Three Runtime Mechanisms

- **Processing**: In the first approach, each application component is executed and displayed on the same device. In the second approach, the first device needs to execute two application components which require high processing capabilities while the second device displays only a video without the need for additional processing resources. In the third approach, the first device needs to render and display the view while the second device needs only to display a video similar to the second mechanism. The video codec used to encode and decode the videos of the captured views plays an important role, especially in the third approach in case the available bandwidth to stream the video from the cloud to the user's device is limited.
- **Software Maintenance**: In general, devices that only need to play videos like D_2 in the second and third approaches do not require a software update and maintenance as for devices that need to execute and render the application locally like devices D_1 and D_2 in the first approach.
- **Disconnection Allowed**: This means that device D_1 can disconnect from D_2 without stopping the application running on it. This is possible in the first and third approaches but not in the second one, since the application is executed on device D_1 and the connection is required to send the image output to device D_2.
- **Multiple Connections**: This means that device D_1 can connect to a new device D_3 at the same time while it is connected to device D_2. This is possible without any limitation in the first and third approaches. In practice, all implementations of the second approach like Miracast and Airplay allow only one connection to the receiver device due to the limited processing capability of device D_1.
- **Scalability**: The first and second approaches are scalable since there are no backend resources required for application execution and rendering during runtime. Only resources for the hosting and delivery of the application are needed.

- **Battery Life**: The battery life is only relevant for devices that are not permanently connected to power like smartphones and tablets. It depends on multiple factors like processing resources needed for executing the runtime function R of each atomic component, for video encoding and decoding, rendering, and display of content. In the first approach, the application needs to execute, render and display content while in the third mechanism the video received from the cloud instance needs to be decoded and displayed which results in similar battery life. In the second mechanism, the battery life on device D_1 is low since it also needs to execute and render the UI of the application component displayed on device D_2.
- **Motion-To-Photon Latency**: In a multiscreen context, Motion-To-Photon latency is the time needed until a user action performed on device D_1 is fully reflected on the display of device D_2. There are different limits for the Motion-To-Photon Latency depending on the use case. For example, in action games, it cannot exceed $20ms$. The first approach has the lowest Motion-to-Photon Latency compared to the other two, since the devices D_1 and D_2 need to exchange only messages with very low latency if both devices are in the same network. In the second approach, device D_1 needs to encode the output of V_2 as video or image stream and send it to device D_2 where it is decoded and displayed. These steps take more time compared to the step for exchanging small messages. The third approach produces the highest latency compared to the first two. The process is similar to the second approach with the exception that the video data is sent over the Internet to device D_2. This means that the connection latency and bandwidth need to be considered in the Motion-To-Photon Latency. Since the bandwidth is limited, video codecs with better compression ratio need to be applied which can also have an impact on the latency. In this case, a balance between latency and video quality needs to be achieved. Figure 4.19 shows the Motion-To-Photon latency of the cloud

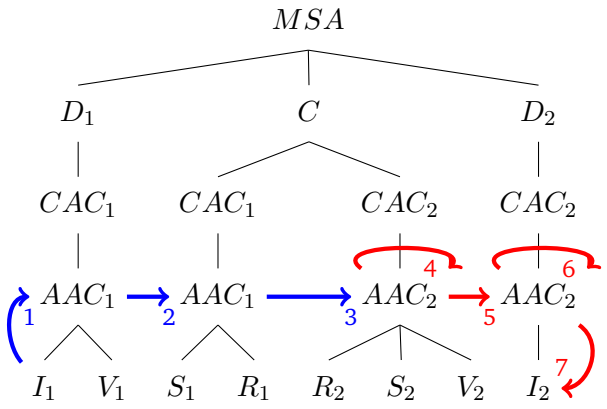


Figure 4.19.: Motion-To-Photon Latency for Cloud Execution Mechanism

execution approach in detail, starting from the user input on device D_1 until

the interaction is reflected on device D_2. The blue arrows (Steps $1..3$) show the flow of control messages while the red arrows (Steps $4..7$) show the flow of video or image data to display on D_2. In Step 1, user inputs are captured in AAC_1 on D_1. In Step 2, the captured inputs are sent over the Internet to AAC_1 running in the cloud instance C. In Step 3, the runtime function R_1 processes the received inputs and interacts with AAC_2 running on the same cloud instance C. In Step 4, the runtime function R_2 of AAC_2 reacts to the data received from AAC_1, updates its view V_2 and captures the UI of AAC_2. The capturing also includes the encoding of the UI output as video or image stream which will be sent over the Internet to AAC_2 on device D_2 in Step 5. In Step 6, the received video or image stream will be decoded and then displayed in Step 7 on device D_2. Therefore, it is recommended to use this mechanism if there are good reasons for this like the use case described in Section 3.1.6 which will be considered in greater detail in Section 5.

- **Offline Capability**: The first and second approaches can be used in offline mode without connecting to the Internet in case all applications and required resources are already installed on the corresponding devices. The third approach requires a connection to the cloud instance, and the offline mode cannot be applied.

In section 6, we will provide a detailed evaluation of the different multiscreen execution approaches under real conditions using the metrics listed above to proof the intermediate evaluation we provided in this section.

4.4.2 Multiscreen Application Framework

The Multiscreen Application Framework (in short Framework) is the second layer of the Multiscreen Application Platform. It consists of different building blocks each of them implementing one of the identified multiscreen features (see Figure 4.15). Not all of the building blocks are mandatory, for example, the framework can provide only one of the *Messaging*, *Eventing* and *Synchronization* components that implement the three approaches *message-driven*, *event-driven* and *state-driven* described in Section 4.3 accordingly if only one of these approaches is desired. Also, the component *UI Capturing & UI Rendering* is only needed if either the *Single Execution Context* or *Cloud Execution* approach described in Section 4.4.1 is selected. Figure 4.20 shows a detailed architecture of the framework layer by considering the three device roles *Sender*, *Receiver*, and *Broker*. Senders are devices like smartphones that discover and launch applications on receivers like TVs. Brokers act as connectors between senders and receivers if direct communication between them is not feasible. In a multiscreen application, at least one device implementing the framework sender components and another device implementing the framework receiver components

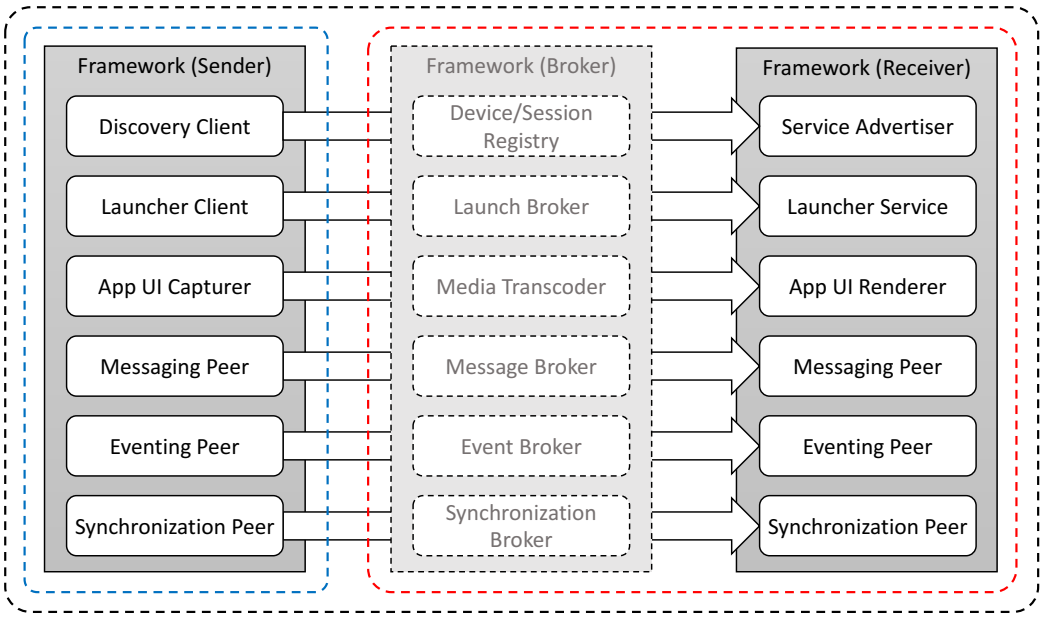


Figure 4.20.: Multiscreen Application Framework

are required. A device can also implement the framework sender and receiver components at the same time. The Framework broker is optional and can run on a dedicated server in the local network, in the cloud or together with the receiver on the same device. The framework components are described below:

- **Discovery/Advertisement**: The *Discovery Client* and *Service Advertiser* are the two counterpart components running on the sender and receiver devices. The *Service Advertiser* is responsible for making the receiver available for senders that run *Discovery Clients* to find receiver devices of interest. If direct discovery is not possible, for example, in case the sender and receiver devices are not in the same network, then a broker entity that may run in the cloud can be used as a central registry for receiver devices that can be easily browsed by senders using certain search criteria.
- **Launch**: The *Launcher Client* and the *Launcher Service* are the two counterpart components running on the sender and receiver devices and allow an application running on the sender to launch another application on the receiver. In some situations, it is not possible to launch an application on the receiver device directly. In this case, a *Launch Broker* is required. This is the case, for example, on most popular mobile platforms like Android and iOS which do not allow an application to launch another one without asking the user.
- **App UI Capturing and Rendering**: The *App UI Capturer* and *App UI Renderer* are the two counterpart components running on the sender and receiver devices and responsible for recording the UI of an application running on the sender device or for rendering the recorded content on the receiver device. In case the sender and receiver devices support different video codecs, then the

Media Transcoder component that runs on the broker will be needed to convert between the source and target video codecs.

- **Messaging**: The *Messaging Peer* components which are provided on the sender and receiver devices are responsible for exchanging data between senders and receivers by implementing the *Message-Driven Approach* approach introduced in the previous section. If a direct communication between the sender and the receiver is not possible, then the *Message Broker* component can be used to forward messages forth and back between the sender and the receiver.
- **Eventing**: Similar to the Messaging component, the *Eventing Peer* components that are provided on the sender and receiver devices implement the *Event-Driven Approach* introduced in the previous section. Each sender or receiver peer can subscribe or publish events to the *Event Broker* that runs either in the cloud or on the receiver device.
- **Synchronization**: The *Synchronization Peer* components on the sender and receiver implement the *Data-Driven Approach* and keep the states of the application components running on the sender and the receiver devices in sync. It applies the concept of *shared object* for synchronization which can be implemented in a decentralized manner by distributing its functionality on each synchronization peer or in a centralized manner by implementing the *shared object* on the *Synchronization Broker*.

4.4.3 Multiscreen Network Protocols

Multiscreen Network Protocols is the third layer of the *Multiscreen Application Platform* which addresses all standards and technologies that are relevant for supporting the components of the framework layer. There is no single protocol supports all multiscreen features at the same time, but instead, a selection of protocols addressing specific functions of the multiscreen platform like discovery, pairing, launch, and communication can be used jointly to build more complex multiscreen functions. The *Open Screen Protocol* [16] which is still work-in-progress at the time of writing this thesis is an open standard developed by the *W3C Second Screen Community Group* [12]. It deals with the specification of network protocols that can be used to implement the *Presentation API* [13] and *Remote Playback API* [14] which are two APIs developed by the *W3C Second Screen Working Group*. The author of this thesis is a founding member of the group and active in the development of the protocol and related APIs. It is not the intention of the group to develop the Open Screen Protocol from scratch, but to evaluate and use existing protocols according to specific multiscreen aspects. Some of these protocols are listed below:

- **Discovery**: SSDP [36] and mDNS/DNS [40] are the two most relevant technologies for discovery in local networks while BLE Discovery [44] is one of the

most relevant discovery technologies to find devices in the range of another device. There are also other proprietary protocols which are not within the scope of this thesis. In the implementation section, we will show how BLE Beacon technology can be used to find nearby devices and launch applications on them.

- **Pairing**: If a device is not able to discover other devices automatically using one of the discovery protocols, then pairing techniques can help to connect the devices manually with the help of the user. QR codes and NFC are two relevant technologies that can be used for this propose.
- **Launch**: DIAL [39] is one of the most relevant protocols for launching applications on remote devices, especially on TVs. There are also other protocols that can be used to launch and control specific services on remote devices like UPnP and Airplay that allow applications to launch and control media rendering (Video, Audio or Image renderer) on TVs instead of launching arbitrary applications.
- **Communication**: HTTP [49], WS [43] and WebRTC [54] are the most relevant protocols that can be applied for the communication between application components running on devices involved in a multiscreen application.
- **App UI Sharing**: Airplay [6] and Miracast [8] are the most popular protocols for capturing and sharing the entire screen or the UI of a specific application in local networks.

4.5 Multiscreen on the Web

In the previous section, we presented a multiscreen application model and discussed various approaches for developing multiscreen applications, followed by an architecture of a multiscreen platform. In this section, we will focus on the applicability of Web technologies like HTML, CSS, JSON, and JavaScript for developing multiscreen applications following the application model we introduced in the previous section. Web technologies have proven to be a cost-effective way to create apps that run on multiple platforms, which is essential for developing multiscreen applications where application components are distributed across multiple devices and platforms. Furthermore, Web technologies are supported on nearly any platform, and some of them like HbbTV, Tizen, WebOS, and Google Cast support only Web technologies for developing applications. Before we introduce the new approach of using Web technologies for developing multiscreen applications, let us have a look at the traditional model for building single-screen Web applications: Traditional Web browsers and Web runtimes are designed to render Web documents hosted on a Web server or are available offline and display the rendered UI to the user on the device's display. Web documents are composed of three main parts: HTML, CSS,

and JavaScript. HTML contains the markup of the content to display, CSS defines how the HTML elements should look like, and JavaScript implements the logic of the application, e.g., listening to user inputs, manipulating the DOM or accessing underlying device APIs. Furthermore, a Web application can request data or perform actions on a server using the *XMLHTTPRequest API (XHR)* or open a bidirectional communication channel to the server using the *WebSocket API*. JSON [135] is used as a web-friendly format for exchanging data between the Web client and the server since it can be easily processed in Web applications without changing its structure. Listing 4.1 shows a simple Web application with the following characteristics:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta name="author" content="L.Bassbouss"/>
5      <title>Simple Web App</title>
6      <style type="text/css">
7        #info {
8          background-color: blue;
9        }
10     </style>
11     <script src="jquery.js"></script>
12     <script type="text/javascript">
13       var ws = new WebSocket("ws://example.com/some/path");
14       ws.onmessage = function(msg){
15         $("#info").text(msg);
16       };
17       addEventListener("deviceorientation", function(e){
18         ws.send(JSON.stringify({alpha: e.alpha, beta: e.beta}));
19       });
20     </script>
21   </head>
22   <body>
23     <div id="info"> </div>
24   </body>
25 </html>
26
```

Listing 4.1: Web Application Example

- It uses the *<meta>* element (*line 4*) to define the *author* metadata. There are also other standardized metadata that allow providers to add more semantic to their applications.
- It uses the *<style>* element (*lines 6-10*) which contains CSS to set the background color of the HTML element with *id=info*.
- It uses the *<script>* element (*line 11*) to load a third party JavaScript library.
- It uses the *<script>* element (*lines 12-20*) that implements the logic of the application using JavaScript: The script opens a *WebSocket* connection to the

server (*line 13*), listens to messages from the server (*line 13*), and updates the text of the info element each time a message is received (*line 15*). It also listens to *deviceorientation* events (*line 17*) and sends the orientation data as JSON string to the server (*line 18*) which creates a user-friendly message and send it back to the client using the same WebSocket connection (*Line 13*).

- It uses the * * HTML element (*Line 23*) to define the view of the application. HTML provides many other elements like *<img>*, *<video>*, *<audio>*, and *<canvas>* that support the development of complex multimedia Web applications with little effort.

As we can see, the Web offers very good tools and building blocks, not only for developing traditional single-screen Web applications but also to develop multiscreen web applications. There is a direct mapping between the three elements (S, V, R) of an atomic application component to Web technologies as shown in Figure 4.21. HTML and CSS can be used to define the view V, JSON to hold the state S and JavaScript to implement the runtime function R. Furthermore, a *Multiscreen JavaScript API* can be provided to allow applications to access multiscreen features without the need to deal with the complexity of the underlying protocols. The author of this thesis published the basic idea of this approach in the paper *Towards a Multi-Screen Application Model for the Web* [17]. Since its publication, the approach of the paper has been improved as Web technologies have been developed further. The solution

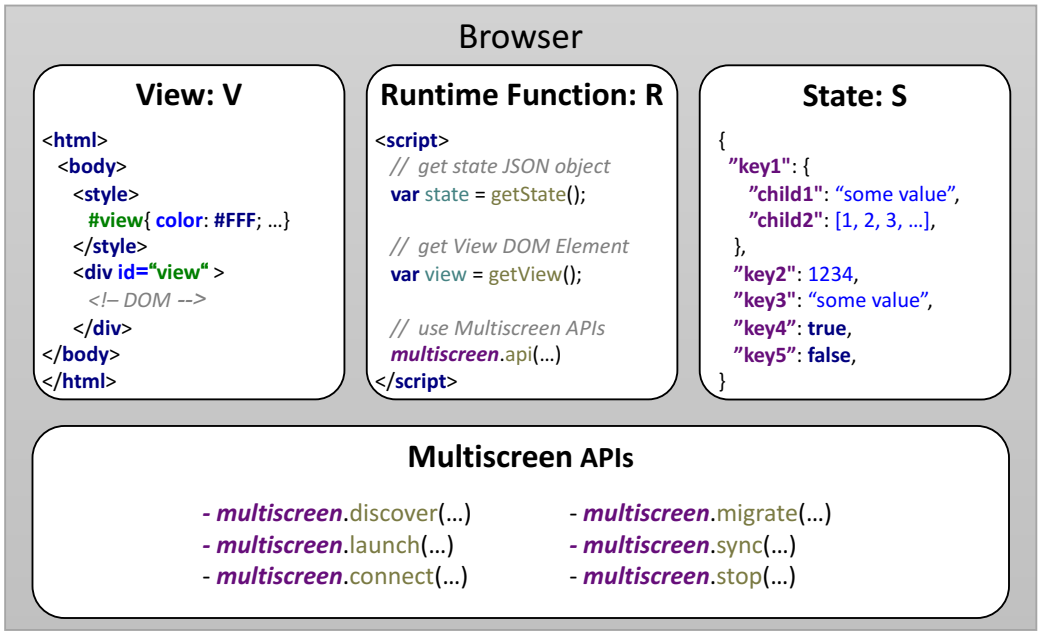


Figure 4.21.: Mapping of the Multiscreen Model to Web Technologies

introduced in [17] allows developers to implement multiscreen web applications in a single document similar to traditional web applications. Listing 4.2 shows a simple multiscreen application document following this approach. The application

can declare itself as multiscreen-capable using the custom <*meta*> element (Line 3). This tells the browser that the web page (also called master page) can receive events when a device (display) is connected or disconnected (Lines $5-11$). The example shows that the application assigns the HTML element with "*id=receiver*" to the connected device (Line 6) and hides it from the master document (Line 7). The element will be visible again in the master document after the device is disconnected (Line 10). The browser will keep the HTML element including its DOM sub-tree in the master document and its mirror element assigned to the connected device in sync. If the user clicks on the *Say Hello* button, the *Hello* text will be added to the HTML element with *id=receiver*. If no device is connected, the text will be displayed on the master page. Otherwise, it will be shown on the connected device. The logic of the application remains unchanged, regardless of whether a device is connected or not.

```
1 <html>
2  <head>
3   <meta name="multiscreen" content="yes"/>
4   <script type="text/javascript">
5    addEventListener("DeviceConnected", function(e){
6     e.device.assign("#receiver");
7     $('#receiver').hide();
8    });
9    addEventListener("DeviceDiconnected", function(e){
10    $('#receiver').show();
11   });
12  </script>
13 </head>
14 <body>
15  <button onclick="$('#receiver').text('Hello');">Say Hello</button>
16  <div id="receiver"></div>
17 </body>
18 </html>
```

Listing 4.2: Multiscreen Web Application Example

The concept introduced in [17] was the first step towards a web-based model for multiscreen applications. Using this model, the development process is nearly the same as for single-screen Web applications. On the other hand, the introduced model has some limitations and cannot be applied to arbitrary multiscreen scenarios. For example, it is difficult to use device APIs and run JavaScript on the target device which are essential features for media-related Web applications. Therefore, the approach introduced in [17] will be extended to consider the multiscreen concepts and approaches presented in Section 4.3 and the multiscreen model tree presented in Section 4.2. The next section introduces a promising HTML technology called *Web Components* that provides relevant building blocks for the development of multiscreen web applications.

4.5.1 Web Components Basics

According to the Multiscreen Model Tree concept, a multiscreen application MSA consists of a set of Composite Application Components CAC_i each of which is associated with a device D_i and consists of a set of Atomic Application Components AAC_{ij}. As discussed before, Web technologies can be used to develop an $AAC = (S, V, R)$: *JSON* can hold the state S, *HTML* and *CSS* can be used to define and describe the layout of view V and the runtime function R can be implemented using *JavaScript*. Since multiple AACs can run in the same CAC on the same device, we need a mechanism that separates the execution of each AAC to avoid conflicts with other components running on the same device. For example, if an AAC uses CSS to define the layout of all * * elements in its DOM, the * * elements of other AACs in the same CAC will be affected as well. The reason for this is the limited scripting capability in *CSS*. This also applies if an AAC needs to find elements in its DOM tree using HTML query selectors. In this case, the DOM elements of other AACs that fulfill the selector will be also considered. The new HTML5 specification *Web Components* addresses these issues and provides a set of promising APIs which allow developers to extend the Web using modular, standards-based, and reusable components that encapsulates the styling and custom behavior with scoping similar programming languages. These are also essential features for developing modular and reusable multiscreen application components. Web components consists of the following four specifications shown in Listing 4.3:

```
<!-- my-component.html -->
<template id="my-template">
 <style>
  h3 {color: blue;}
 </style>
 <div> <h3>This is a simple Web Component</h3> </div>
</template>
<script>
 class MyComponent extends HTMLElement {
  constructor() {/*...*/}
  static get observedAttributes() {/*...*/}
  attributeChangedCallback(attrName, oldValue, newValue) {/*...*/}
  disconnectedCallback() {/*...*/}
  connectedCallback() {
   var template = document.querySelector('#my-template').content;
   var shadow = this.attachShadow({mode: 'open'});
   shadow.appendChild(document.importNode(template, true));
  }
 }
 customElements.define('my-component', MyComponent);
</script>

<!-- index.html -->
<link rel="import" href="my-component.html">
```

```
25 <my-component></my-component>
```

Listing 4.3: Multiscreen Application Example

Custom Elements The *Custom Elements* specification [136] is still under development as part of the *W3C Web Platform Working Group* [137] and has the status "Working Draft" at the time of writing this thesis. It allows Web developers to define their own fully-featured DOM elements. The example depicted in Listing 4.3 showcases all *Web Components* features including *Custom Elements*. The first step is the definition of the custom element class $MyComponent$ (*Lines 9-19*). This class must always inherit from $HTMLElement$ or any subclass of it. The $constructor$ (*Line 10*) supports initializing the newly created instance. It is also possible to observe the value of an attribute by overriding the $attributeChangedCallback(...)$ method. Only attributes returned by the method $observedAttributes()$ can be observed. Furthermore, the methods $connectedCallback()$ and $disconnectedCallback()$ can be overridden to get notified after the element is appended or removed from the DOM. In addition to the definition of the element class, it must be registered using the function $customElements.define()$ (*Line 20*). After this, the new element *<my-component>* can be used like any other HTML element (Line 25). It can be also instantiated and added to the DOM using JavaScript: $appendChild(new\ MyComponent())$.

Shadow DOM Similar to other Web Components specifications, *Shadow DOM* [138] is a living standard under development in the *W3C Web Platform Working Group*. It provides a way to encapsulate the *DOM* and *CSS* in a Web Component. It separates the DOM of a custom or literal HTML element from the DOM of the main document. This avoids CSS styling conflicts especially on large pages or if the application uses third-party Web components. The Web component example depicted in Listing 4.3 uses *Shadow DOM*: The method $attachShadow()$ (*Line 16*) creates a *Shadow DOM* and attaches it to the defined custom element. The same DOM manipulation methods can also be used to manipulate the shadow DOM. For example, the method $appendChild()$ (*Line 17*) can be used to add elements to the shadow DOM.

HTML Templates *HTML Templates* [139] is also developed in the *W3C Web Platform Working Group* and allows developers to write markup templates that are not displayed on the rendered page. Templates are defined in the HTML *<template>* element and can be reused multiple times in the application by cloning the content of the template element and appending it to any HTML element. In Web components, the DOM and CSS styling can be defined through HTML templates (*Lines 2-7*), and each time a custom element is appended to the main document, the content of the template will be cloned and appended to the shadow DOM (*Lines 15* and *17*).

HTML Imports The implementation of a Web component that includes the definition of *Custom Elements* using *Shadow DOM* and *HTML Templates* can be kept in a separate HTML file and be reused in other HTML documents. *HTML Imports* provide a way to do this using the new *<link>* type *import*. In Listing 4.3, the main page *index.html* imports the Web component HTML file *my-component.html* (*Line 24*) in order to use the new element *<my-component>*. As we can see in the main document, we can import the Web component and use it by just two lines of code.

4.5.2 Web Components for Multiscreen

In this section, we will investigate the adoption of Web components for developing multiscreen applications following the concepts and models we presented in this chapter. The main motivation for applying Web components in the multiscreen domain is that this technology has many features in common with *Composite Application Components* and *Atomic Application Components*, e.g., the modular design, reusability, easy instantiation and ability for migration between Web documents. In other words, *Composite Application Components* and *Atomic Application Components* can be considered as *Web Components* with extended multiscreen functionalities that are not necessarily relevant for single-screen Web applications. Figure 4.22 shows the UML class diagram which corresponds to the Multiscreen Model Tree using Web Components. As we can see, CAC and AAC are two abstract Web component

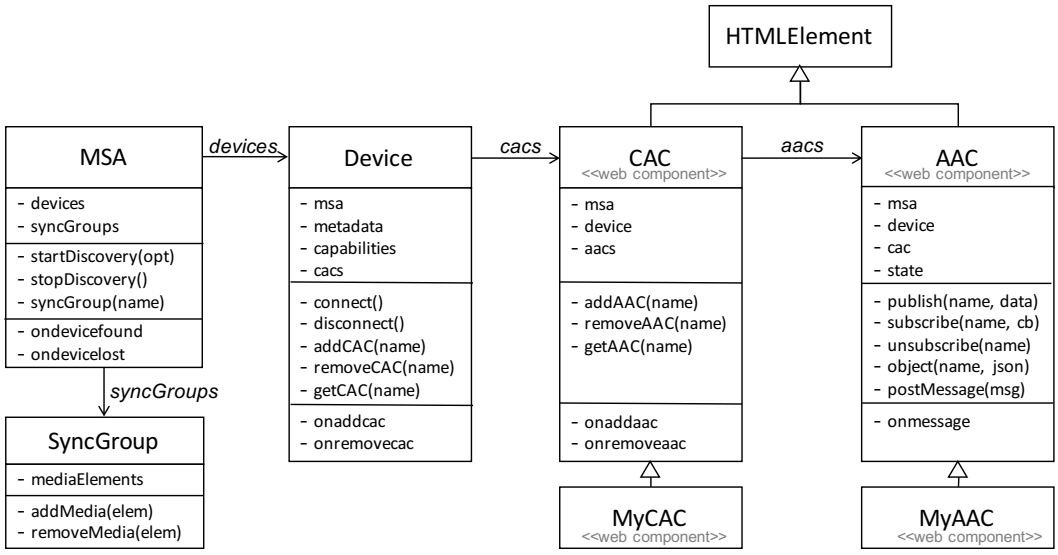


Figure 4.22.: Web Components for Multiscreen (UML Class Diagram)

classes that inherit from $HTMLElement$ and implement common functions for all composite and atomic application Web components. The classes $MyCAC$ and $MyAAC$ are concrete implementations of composite and atomic application components. Each concrete implementation of a composite application component must inherit from the generic CAC Web component class, and each atomic application

component must inherit from the generic AAC Web component class. The four main classes depicted in the UML diagram are described below:

- ***MSA***: All instances of the *MSA* class are representatives of the multiscreen application on each device. The *MSA* class provides the methods $startDiscovery()$ and $stopDiscovery()$ to start and stop discovery of devices independent of the used technology. The events $devicefound$ and $devicelost$ will be triggered each time a new device is discovered, or an existing device disappears. The event data contains the discovered or disappeared $Device$ instance. Furthermore, each *MSA* instance holds a list of devices on which the multiscreen application is currently running.
- ***Device***: The *Device* class represents device instances which are discovered by the application or are currently running application components. The list of device instances will be kept in sync with the actual physical devices that run the multiscreen application. A device provides metadata and information about its capabilities which can be used for device filtering. The metadata contains device name, manufacturer and other relevant information about the device. A device also offers the methods $connect()$ and $disconnect()$ that enable connecting a new device to the multiscreen application or disconnecting an existing device. The list *devices* in the *MSA* class will be updated accordingly. After connecting to a device, the methods $addCAC()$ and $removeCAC()$ can be used to add or remove composite components to or from the device. Furthermore, any component can monitor if a composite application component is added or removed from the launched application by subscribing to the events $onaddcac$ and $onremovecac$.
- ***CAC***: The *CAC* class represents composite application component instances and must always inherit from $HTMLElement$ since a CAC is always a Web component. The *CAC* class itself is an abstract class, and each concrete implementation must inherit from it such as the $MyCAC$ class depicted in the UML diagram. A CAC instance holds references to the MSA and the $Device$ instances on which the component is running. Before a device can be used, the application must connect to it via the $connect()$ method. Similarly, the $disconnect()$ method allows an application to disconnect from a device with the option to keep or terminate the application. Through the methods $addAAC()$ and $removeAAC()$, a new AAC can be added, or an existing AAC can be removed which will trigger the events $onaddaac$ or $onremoveaac$. Both methods can only be used if the application is currently connected to the device.
- ***AAC***: The *AAC* class represents atomic application component instances and must always inherit from $HTMLElement$ since an AAC is always a Web component similar to CACs. An AAC instance holds also references to the MSA and the $Device$ instances on which the component is running and to the parent CAC instance. Furthermore, the AAC class offers methods related

to the supported multiscreen approaches presented in Section Multiscreen Application Concepts and Approaches. In case the Message-Driven Approach is supported, an AAC can receive messages sent by other AACs by listening to $onmessage$ events which contain the event data (in $event.data$) and the sender AAC ($event.source$). The counterpart method for the $onmessage$ event is $postMessage()$ which allows to send a message to a specific AAC. If the Event-Driven Approach is supported, then any AAC can publish or subscribe to events from specific types using the $publish()$ and $subscribe()$ methods. Finally, if the Data-Driven Approach is supported, then the method $object()$ which creates or connects to a named shared object can be used. This method returns an object with a structure and interfaces similar to JSON. Any changes to the content of the shared object will be synchronized across all components on all devices that hold a copy of the shared object with the same name. An AAC can observe changes to any property of the shared object by using the $observe(path, listener)$ function which takes the JSON $path$ as input of the property under consideration and a $listener$ function that will be triggered each time the value of the corresponding property is updated. At least one of the three approaches must be implemented in order to allow the application components to interact with each other.

To illustrate the usage of Web Components for developing multiscreen applications following the concept described above, let us consider a *Multiscreen Slides* application such as *Google Slides* as an example. Figure 4.23 shows useful combinations of the following four *AAC*s on devices like laptop, TV, projector, and smartphone:

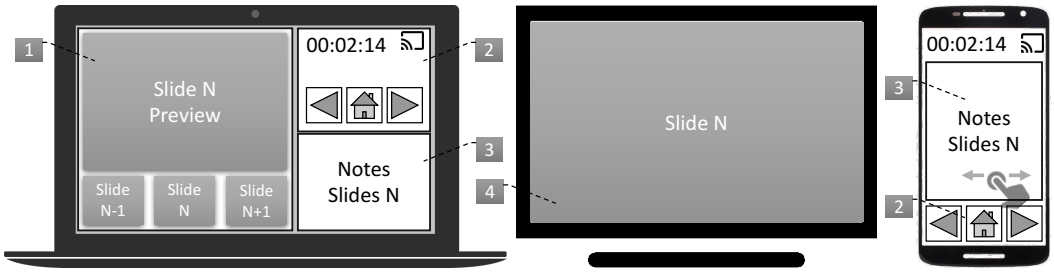


Figure 4.23.: Multiscreen Slides

1. ***Slide Preview AAC***: provides a preview of the current slide and a thumbnail view of all other slides.
2. ***Slide Control AAC***: provides UI elements for selecting the current slide like buttons for switching to the previous, next or first slide. Other UI elements to open or load the presentation slides may also be added to this component.
3. ***Slide Note AAC***: shows the notes of the current slide if they exist.
4. ***Slide Show AAC***: shows the current slide on the presentation display.

In this example, the laptop and the smartphone act as a presenter and the TV as display for the slides. Due to its small screen size, the smartphone shows only the *Slide Note AAC* and *Slide Control AAC* while the laptop shows also the *Slide Preview AAC* in addition to these two *AAC*s. The composite application components that cover the three combinations of *AAC*s are listed below:

1. ***Slide Display CAC*** for TV or projector: is a container for the *Slide Show AAC*.
2. ***Slide Presenter CAC*** for laptop or PC: is a container for *Slide Control AAC*, *Slide Note AAC* and *Slide Show AAC*.
3. ***Slide Presenter CAC*** for smartphone: is a container for *Slide Control AAC* and *Slide Note AAC*. Furthermore, the control AAC adapts to the input capabilities of the smartphone and allows the user to switch to the previous or next slide by swiping on the touch screen to the left or to the right.

It is important to mention that multiple composite component instances can be launched simultaneously on multiple devices. For example, the *Slide Display CAC* can be launched on multiple presentation displays. Also, the *Slide Presenter CAC* can be launched on multiple laptops or smartphones, if multiple users are collaborating on a single presentation.
After identifying the atomic and composite application components of the multiscreen slides application, it is important to select the best suitable of the multiscreen approaches presented in Section 4.3. In case the Message-Driven Approach is selected, each application component must ensure that every single update is reflected on all other components by exchanging messages. The complexity of using this approach increases with the growing number of connected devices since each component must maintain a list of remote components on which the $postMessage()$ method is called in order to send the messages. The Event-Driven Approach reduces the complexity of the Message-Driven Approach since the components do not need to know about each other, but only a set of events need to be defined. The only limitation of the Event-Driven Approach is that new devices that join the application must ensure that the components are initialized properly by asking already running components about their current states. This issue can be solved by using the Data-Driven Approach where a shared object is initialized automatically if another component already created the object. Therefore, this approach is selected for developing the multiscreen slides application. The structure of the shared object is kept simple in this example application. It consists of the two properties $slides$, an array of JSON objects where each item in the array represents a slide, and $currSlide$ that indicates the number of the current slide (index in the slides array). Each JSON object in the $slides$ array consists of the two properties $content$ and $notes$ which hold the content and notes of the corresponding slide. Listings 4.4 and 4.5 show the Web components implementation of the two atomic components *Slide Control AAC* and *Slide Show AAC*. The complete implementations of all Atomic and Composite

Web Components of the Multiscreen Slides application are provided in Appendix B.1.

```
/* Slide Control AAC Web Component: aac-control.html*/
<template id="aac-control">
 <style>
  /* styles for the control AAC */
 </style>
 <div>
  <button id="open-btn">Open Slides</button><br>
  <button id="prev-btn">Previous Slide</button><br>
  <button id="next-btn">Next Slide</button><br>
 </div>
</template>

<script>
 class AACControl extends AAC {
  connectedCallback() {
   var template = document.querySelector('#aac-control').content;
   var shadow = this.attachShadow({mode: 'open'});
   shadow.appendChild(document.importNode(template, true));
   var openBtn = shadow.querySelector("#open-btn");
   var prevBtn = shadow.querySelector("#prev-btn");
   var nextBtn = shadow.querySelector("#next-btn");
   this.msa.object("state",{
    currSlide: 0,
    slides: []
   }).then(function(state){
    openBtn.onclick = function(){
     this.loadSlides().then(function(slides){
      state.slides = slides;
     });
    }
    prevBtn.onclick = function(){
     state.currSlide > 0 && state.currSlide--;
    }
    nextBtn.onclick = function(){
     state.currSlide < slides.length-1 && state.currSlide++;
    }
   });
  }
  loadSlides(){
   /* load slides from somewhere */
  }
 }
 customElements.define('aac-control', AACControl);
</script>
```

Listing 4.4: Multiscreen Slides using Data-Driven Approach: Slide Control AAC Web Component

As we can see in the Control AAC Web Component in Listing 4.4, it consists of a template part where the UI and styles of the component are declared (*Lines 2-11*), a JavaScript implementation of the $AACControl$ component which inherits from the AAC generic class (*Lines 13-41*) and a registration of the $AACControl$ class as a Web Component under the name $aac{-}control$ (*Line 42*). This component can be instantiated in any composite component either programmatically in JavaScript using the $AACControl()$ constructor or declaratively using the custom HTML tag <*aac-control*>. The main part of the implementation is the $connectedCallback()$ function which will be triggered after the component is added to the *DOM*. After the HTML elements from the template are added to the shadow DOM of the component, the shared object $state$ will be created and initialized if it does not exist yet. If a shared object with the same name $state$ was already created by another component on the same or another device, then a copy will be created and kept in sync with any other shared objects of the same name. For example, the Slide Show AAC component in Listing 4.5 creates also a shared object with the same name (*Lines 12-15*). In order to switch to the next slide, the click handler of the $nextBtn$ only needs to increment the value of $state.currSlide$ in the shared object. Other components observing the same property will be notified after each change to that property. For example, the Slide Show AAC component in Listing 4.5 observes changes to any property of the shared object $state$ and updates the UI accordingly (*Lines 16-19*).

```
1 /* Slide Show AAC Web Component: aac-show.html */
2 <template id="aac-show">
3   <p id="slide"></p>
4 </template>
5 <script>
6  class AACShow extends AAC {
7   connectedCallback() {
8    ...
9    var slideEl = shadow.querySelector("#slide");
10   this.msa.object("state",{
11    currSlide: 0,
12    slides: []
13   }).then(function(state){
14    state.observe("*",function(newVal, oldVal, path){
15     var slide = state.slides[state.currSlide];
16     slideEl.innerHTML = slide && slide.content? slide.content:"";
17    });
18   });
19  }
20 }
21 customElements.define('aac-show', AACShow);
22 </script>
```

Listing 4.5: Multiscreen Slides using the Data-Driven Approach: The Slide Show AAC Web Component

After the atomic application components are implemented (each in a separate HTML document), the composite application components can now be developed also as Web components by including the atomic components using HTML imports. Usually, the composite components are containers for atomic components with additional logic for layouting and positioning of these atomic components. Furthermore, a composite component is the right place for implementing the distribution logic of the application. In the multiscreen slides example, the *Slide Presenter CAC* is usually the component that is launched manually by the user on his device. It can discover presentation displays and launch the *Slide Display CAC* on one of the discovered displays selected by the user. Listing 4.6 shows the implementation of the *CAC Presenter Component* which imports three *AAC* components (*Lines 2-4*). In this example, we use the declarative method for creating the *AAC* instances (*Lines 11-13*). It is also possible to use the scripting method, but it requires more line of codes to create an *AAC* instance and append it to the DOM. Furthermore, the UI of the *Presenter CAC* provides a button (*Line 10*) which can be used to discover devices and to launch the *Display CAC* on one of them (*Lines 22-42*).

```
1 /* Slide Presenter CAC Web Component: cac-presenter.html */
2 <link rel="import" href="aac-preview.html">
3 <link rel="import" href="aac-notes.html">
4 <link rel="import" href="aac-control.html">
5 <template id="cac-presenter">
6  <style>
7   /* styles for positioning and styling the AACs */
8  </style>
9  <div>
10   <button id="present-btn">Present</button>
11   <aac-preview></aac-preview>
12   <aac-notes></aac-notes>
13   <aac-control></aac-control>
14  </div>
15 </template>
16
17 <script>
18  class CACPresenter extends CAC {
19   connectedCallback() {
20    ...
21    var self = this;
22    presentBtn.onclick = function(){
23     self.discoverFirstDevice().then(function(device){
24      device.launch("cac-display");
25     }).catch(function(err){
26      /* no device found */
27     });
28    }
29   }
30   discoverFirstDevice() {
31    var self = this;
```

```
32    return new Promise(function(resolve, reject){
33     self.ondevicefound = function(evt){
34      self.stopDiscovery();
35      resolve(evt.device);
36     }
37     setTimeout(function(){
38      self.stopDiscovery();
39      reject(new Error("No device found"));
40     },5000);
41    });
42   }
43  }
44  customElements.define('cac-presenter', CACPresenter);
45 </script>
```

Listing 4.6: Multiscreen Slides using Data-Driven Approach: Slide Presenter CAC Web Component

4.6 Implementation

In the previous section we introduced the application of the Web Components technology for developing multiscreen applications following the concept of atomic and composite application components. Now we focus on the implementation of selected components of the Multiscreen Application Architecture presented in Section 4.4.

4.6.1 Discovery and Launch

Discovery and Launch are two essential features in a multiscreen environment. Discovery enables an application component to find relevant devices while launch starts an application component on a discovered device. The methods of the MSA and $Device$ classes depicted in the UML diagram of Figure 4.22 are bound to the discovery and launch functions of the underlying multiscreen application framework. In this section, we will discuss and explain the implementation of discovery and launch using state-of-the-art technologies and standards. We will focus on Web technologies and consider the browser as application runtime.

To trigger the discovery, an atomic application component calls $startDiscovery()$ which forwards the request to the discovery engine of the underlying multiscreen framework. The discovery request should contain at least the $launcherUrl$ pointing to the composite application component to launch plus other optional filter parameters like device type and required capabilities. The result of the discovery process is a list of devices where each item in the list consists of the parameters $friendlyName$, $deviceId$, and $launcherEndpoint$. Other device metadata like manufacturer and

device capabilities can also be included in the device description. It is also important to note that receiver devices should make their multiscreen services discoverable by other devices, e.g. through service advertising in the local network or by registering them in a service repository. In this thesis, we will focus on the implementation of the following service discovery methods: 1) context-based lookup in a device registry, 2) discovery of devices in the same network and 3) discovery of nearby devices. Each of these methods will be explained in detail below.

Context-based lookup in a device registry The context-based lookup process in a device registry is shown in the sequence diagram in Figure 4.24. The context

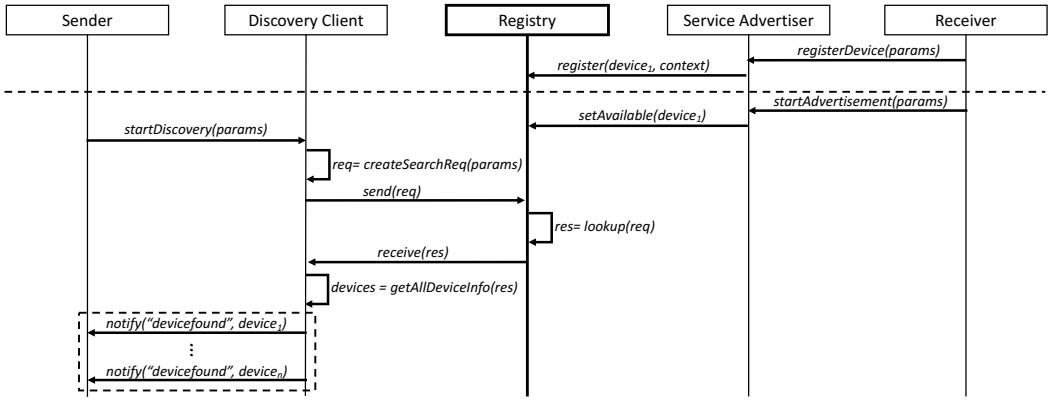


Figure 4.24.: Context based lookup in a device registry

is defined as a set of properties or key-value pairs which are used to register the device in a central registry. In most situations, it is bound to a user account which supports the discovery of devices belonging to the user after login. The $Registry$ stores also device metadata and capabilities which are used to filter devices during lookup. The context can be extended, for example, to facilitate the discovery of devices for a group of persons. Platforms like Apple TV and Amazon Fire TV use this concept by assigning devices to a user account. The device registration step needs to be done only one time until the device is deregistered by removing its entry from the registry. A registered device is not necessarily always accessible, as it can be switched off at any time. Therefore, each time the receiver is switched on, it advertises itself as "available" in the registry. The sender can start discovery by sending a search request to the $Registry$ which queries the database for devices that fulfill the request and send them back to the sender. The sender triggers the $devicefound$ event for each device in the list. Newly available devices will also be returned until the $stopDiscovery()$ is called. After discovery is completed, in the next step the sender can use the $device.launcherEndpoint$ of the selected device to launch the composite component.
The implementation of this discovery method is straightforward: The JSON format can be used for message exchange between the $Sender$ or $Receiver$ and the $Registry$, and $WebSocket$ or $HTTP$ can be used as transport protocols.

Discover devices in the same network The context-based lookup has the disadvantage that it requires a $Registry$ as a central entity to manage the devices and the user needs to perform additional steps, for example, log in on all devices. Furthermore, each manufacturer uses its own registry, and it is difficult to standardize a neutral central entity that works across providers. *Network Service Discovery* solves these issues in case the sender and the receiver devices are connected to the same network. The idea behind this method is depicted in the sequence diagram in Figure 4.25. After the receiver device is turned on, it advertises its multiscreen services in the net-

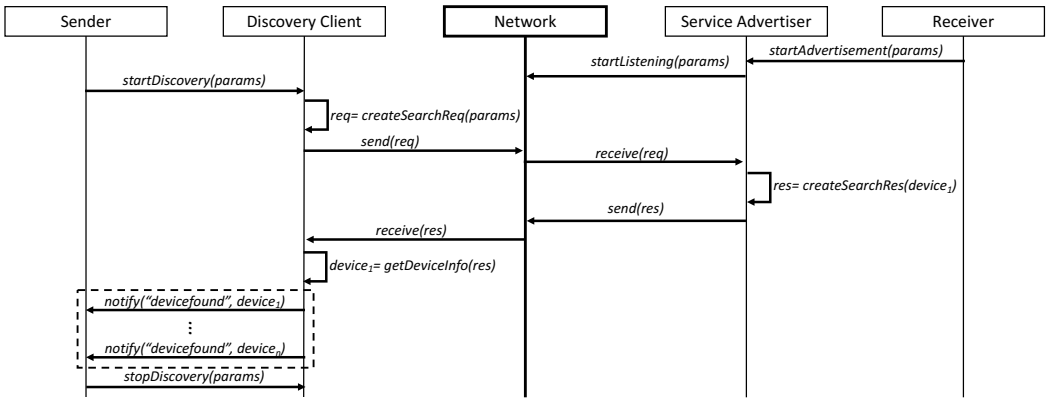


Figure 4.25.: Discover devices in the same network

work by using *UDP multicast* which contains basic information about the device and service endpoints such as the $launcherEndpoint$ in our case. Furthermore, the receiver listens to search requests sent to the multicast address after $startDiscovery()$ is called on the sender. The sender receives responses from all devices in the network that fulfill the request and triggers the $devicefound$ event for each device found until $stopDiscovery()$ is called. $SSDP$ and $mDNS/DNS-SD$ are the most two relevant protocols that support network service discovery and are supported on most TV and mobile platforms. The author of this thesis published an implementation of the *SSDP* protocol for *Node.js* [140] as well as for *Android* and *iOS* as part of the HbbTV Cordova Plugin [141]. As we can see in both discovery methods, the sender will get a list of devices with $friendlyName$ and $launcherEndpoint$ for each discovered device. The $friendlyName$ will be presented to the user in the device selection dialog while the $launcherEndpoint$ is used to launch the application on the selected receiver device. There are two possible implementations for "launch": 1) The receiver offers a *Launcher Service* with an *HTTP API* behind the $launcherEndpoint$ (for example a REST API). The *Launcher Client* running on the sender sends a request to the $launcherEndpoint$ with all relevant data in the HTTP body to launch the application; 2) If direct communication between the sender and the receiver is not possible, for example, if both devices are not connected to the same network, then a *Launcher Proxy* that runs on a central server can be used to bridge the requests between the sender and the receiver. In this case, the *Launcher Service* running on the receiver must establish a bi-directional communication channel to the *Launcher*

Proxy and waits for launch requests pushed to this channel. *WebSocket* supports this type of communication between the *Launcher Client* or the *Launcher Service* and the *Launcher Proxy*.

Discover devices nearby The network service discovery approach is mostly relevant for networked devices that are always connected to a power supply and can run services in the background all the time. In most cases these are TVs or streaming devices like Apple TV, Fire TV, and Chromecast. The sender devices, e.g., smartphones and tablets, do not need to run any service in the background, but only start discovery on demand when the application is running in the foreground. The question is how we can enable the discovery of companion devices from TV applications. In this case, the TV plays the role of the sender and the companion device the role of the receiver. It is not recommended to run a launcher service in the background on the smartphone and use network service discovery to advertise the service for many reasons: 1) Many mobile platforms like *iOS* provides limited support for running services in the background, especially when it comes to services that accept requests from other networked devices, 2) running services in the background will have impact on the battery life, and 3) running background services on personal devices will have impact on privacy and security by opening the door for attackers especially in open networks. For this reason, in this section we will provide a solution based on the *BLE beacon* technology to support the discovery of nearby devices from the TV by taking advantage of existing discovery approaches and considering the limitations above. The author published this approach in [18]. The details of the approach are provided in the next section.

Discovery and Launch using iBeacon

In order to discover companion devices like smartphones or tablets from an application running on the TV, it is necessary to use an appropriate technology that finds only devices within a specific range of the TV and without affecting other aspects like usability, battery life, privacy, and security. *Bluetooth Low Energy (BLE)* [44] is one of the relevant technologies worth further investigation, as only devices within the range of the BLE transmitter which receive the BLE signal are considered in the discovery. Furthermore, the distance between the BLE transmitter and the receiver device can be estimated from the measured signal strength. This information can then be used, for example, to sort the list of discovered devices presented to the user by distance.

"*Also, the flow for remotely launching companion applications is from a usability perspective, not the same as for launching TV applications. Putting an application on a companion device in the foreground without asking the user is an annoying ex-*

perience for the user. Most mobile platforms enable this feature only in combination with user interaction, for example, when the user starts a new application by clicking on a button in the current application or from a notification. On iOS, there are two types of notifications: local and remote notifications. The end-user does not see any difference between them since, they differ only in the way how they are triggered: Local notifications are triggered by applications running in the background on the same device while remote notifications are sent via the Apple Push Notification service (APNs) [142]. This means that if an application is not running at all (neither in the foreground nor in the background), the user can be notified only through remote notifications. One option to wake-up and launch a not running iOS application in the background is by using iBeacon, a technology that extends Location Services introduced in iOS7. iBeacon uses a Bluetooth Low Energy (BLE) signal which can be received by nearly all iOS devices. Any device supporting BLE can be turned into an iBeacon transmitter and alert applications on iOS devices nearby. In general, iBeacon transmitters are tiny and cheap sensors that can run up to 2+ years with a single coin battery depending on how frequent they broadcast information. The main usage area of iBeacon is for location-based services: Apps will be alerted when the user approaches or leaves a region with an iBeacon. While a beacon is in range of an iOS device, apps can also monitor the relative distance to the beacon. If the application was not running while the user crosses (enters or leaves) the region of a beacon, the iOS device wakes up the application and launches it in the background for 10 seconds only (iOS limitation to save battery). During this time, the application can respond to changes in the user position and may request to show a local notification, through which the user can bring the application to the foreground. Based on this, we will propose some ideas for a user-friendly remote launch mechanism of TV companion applications using iBeacon and push notification technologies. We will limit this solution to iOS devices that support iBeacon and consider Apple's application development guidelines to ensure best user experience. Nevertheless, the concept can be adapted easily to other devices and platforms that support BLE, e.g., Android. Unlike on iOS which is the only platform that provides native iBeacon support, the iBeacon functionality needs to be implemented on application level for other platforms.
As mentioned before, the iBeacon protocol uses BLE technology to transmit information in a specific interval, for example, every second. Besides the BLE packet headers and Apple's static prefix, an iBeacon message consists of the following values:

- *Proximity UUID: A 128-bit value that uniquely identifies one or more beacons as a certain type or from a certain organization.*
- *Major: A 16-bit unsigned integer that can be used to group related beacons that have the same proximity UUID.*
- *Minor: A 16-bit unsigned integer that differentiates beacons with the same proximity UUID and major value.*

iOS applications can use the iBeacon API introduced in iOS7 for registering a beacon's region using the proximity UUID, major and minor parameters described above. If a device crosses the boundaries of a registered beacon's region, the application will be notified on entering or leaving that region. The proximity UUID is mandatory for registering a Beacon region while major and minor are optional. The application provider needs to choose a value for the proximity UUID and use it in all beacons as well as in the iOS application. This means that the proximity UUID is a static value in the context of a specific application or organization. Major and minor values can be used to differentiate between different locations or places for the same application or organization. iBeacon seems to be a promising technology not only for location-based services, but also for launching companion applications in a multiscreen environment if the new generation of TVs and streaming devices are equipped with BLE sensors and act as beacon transmitters. The main advantage of this approach is that the TV will be able to wake-up companion applications only on devices belonging to viewers sitting in front of the TV independent if they are connected to the local or mobile carrier network. Unlike the traditional usage of iBeacon where the proximity UUID is static and known for a specific application, a more dynamic behavior using different values for the proximity UUID on different TV sets is required in the multiscreen domain: If the TV manufacturer uses a unique proximity UUID, the companion application will always be notified when the user crosses the beacon region of any TV from the same manufacturer. In case the TV sets transmit different proximity UUIDs, it will be possible to notify companion applications associated with a specific TV. Furthermore, it is possible for a companion application to subscribe to different beacon regions at the same time and therefore to get notified by different TV sets, i.e., if the user has more than one TV at home. Figure 4.26 shows an example with two TV sets that transmit different proximity UUIDs. Companion Device 2 is registered for both proximity UUID1 and proximity UUID2 and can be notified from both TV1 and TV2. The other two companion devices can only be notified from one TV set. Since there is no unique and known proximity UUID to be used in the TV companion application, we need a mechanism to generate and exchange proximity UUIDs between the TV and the companion application. UUIDs can be randomly generated and stored on the TV without any user interaction. The probability of collision with UUIDs used in other applications is almost zero since proximity UUIDs are 128-bit long. The best way to exchange the generated UUID is during first connection (setup phase) of the companion application with the TV. Figure 4.27 illustrates the steps needed for the creation and exchange of the proximity UUID between TV and companion applications. After this, each time the user turns on the TV or enters the TV's beacon region (e.g., living room) while the TV is on, the companion application will be woken up and launched in the background for approximately 10 seconds. The same applies if the user turns off the TV or leaves the TV's beacon region. In both cases, the companion application connects to a signaling server (e.g., maintained by the TV manufacturer) when running in the background and requests to update its availability in the TV's beacon region by sending the proximity UUID and the device

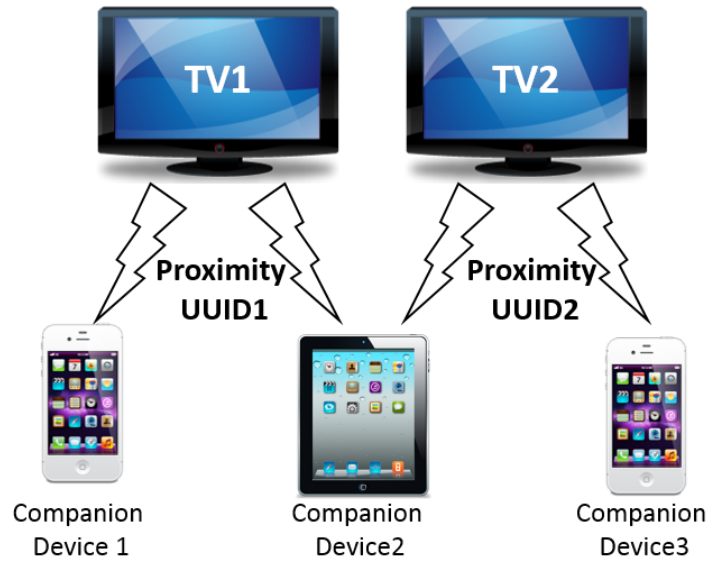


Figure 4.26.: Example with two TV sets and three companion devices

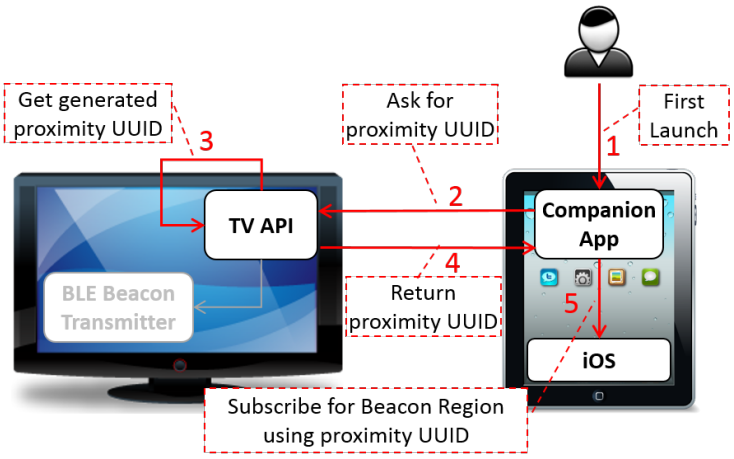


Figure 4.27.: Creation and Exchange of proximity UUID

token (we suppose that the companion application already requested a device token from the Apple Push Notification Service). As depicted in Figure 4.28, the signaling server maintains a table for device availability and offers a lookup function to find devices (identified by the device token) in range of a specific beacon (identified by the proximity UUID). On the other hand, if a TV Application (Hybrid or Smart Application) provides multiscreen support and needs to launch a companion application, it uses a specific TV API for this purpose (Figure 4.28 - step 2). The TV sends its proximity UUID and other application-specific information to the Signaling Server (Figure 4.28 - step 3). The signaling server searches in the table for all devices in range of the beacon with the received proximity UUID (Figure 4.28 - step 4) and sends a request to the APN service using the tokens of the devices from the previous step. The APN service sends push notifications containing all information necessary to launch the companion application on each device found (Figure 4.28 - step 6). If the user clicks on the push notification,

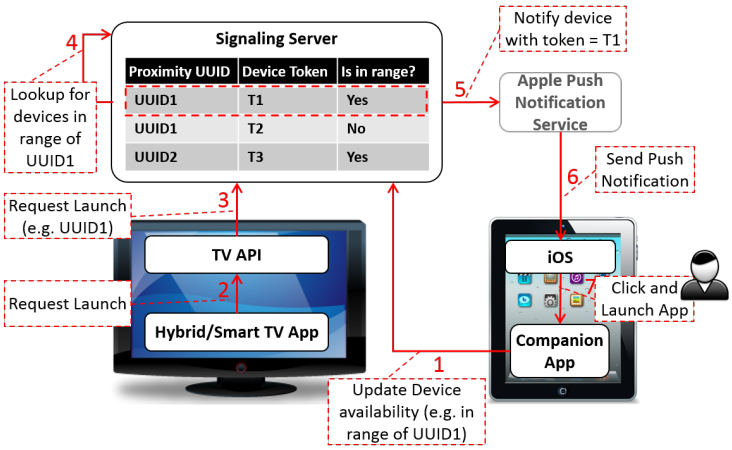


Figure 4.28.: Launch a Companion Application from a TV Application

the companion application will be launched in the foreground, and the notification data will be passed to it (Figure 4.28 - step 7). Remote push notifications are necessary in this scenario because local notifications are only possible when the app is running in the background at that moment which is not necessarily the case. Though apps are woken up through iBeacons, they only run for 10 seconds and are then terminated by the operating system. Moreover, the OS will only wake up apps when entering or exiting a region of a beacon. For this reason, we cannot assume that the Companion App is running at this time and available to send a local notification. Therefore, it is necessary to use remote push notifications via APNs.

Figure 4.28 shows the flow for notifying and launching the companion application provided by the TV manufacturer from a hybrid or smart TV application provided by a broadcaster or a third party provider. However, our goal is to launch the companion application related to the TV application and not the manufacturer companion application. There are different options to achieve this goal depending on the kind of the companion application to launch if it is a hosted web application or a native iOS application. We will focus in this work on hosted web applications. As mentioned above, The TV application passes information about the application to launch on the companion device in step 2. It includes a URL of the hosted Web application to launch and will be passed to the companion device through all steps in Figure 4.28 until the user clicks on the notification. The TV companion application will be launched in the foreground and can retrieve the URL of the hosted companion web application from the launch information passed to it. Finally, the TV companion application opens the hosted companion application in a Web View (UIWebView), a kind of integrated web browser for displaying web content in iOS applications. Now the TV application and the hosted companion web application can collaborate and synchronize content between each other by using an appropriate communication mechanism" [18].

4.6.2 Communication and Synchronization

After an application is launched on a receiver device, a communication channel can be established between the sender and receiver components. In Section 4.3 we introduced the three available approaches, message-driven, event-driven, and data-driven, for developing multiscreen applications. All three approaches rely on a communication layer between the multiscreen components. The message-driven approach is the easiest to implement since it can be mapped directly to the underlying communication layer. The other two approaches require an intermediate layer between the *Multiscreen API*, and the communication layer.
In a first step, let us consider the implementation options for the establishment of a communication channel between two multiscreen components running on two different devices. Communication between two components running on the same device is also essential but its implementation is straightforward, and therefore our focus will be only on inter-device communication. In both cases, all APIs provided to the application (for all three approaches) should abstract from the underlying communication protocol. As depicted in Figure 4.29, there are two ways for two application components to establish a communication channel:

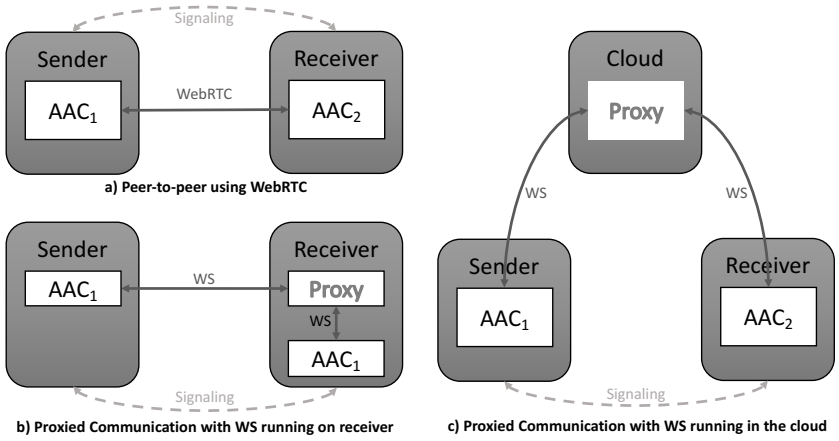


Figure 4.29.: Direct VS. Indirect Communication

- **Direct communication**: both application components establish a peer-to-peer communication channel without the need for a third entity (intermediate server or proxy). In Web environments, *WebRTC* [54] is the most appropriate protocol for this kind of communication. It is based on *UDP* but at the same time offers reliable communication. It is supported in all modern browsers for mobile and desktop platforms, but its support on TV devices is still limited. Although *WebRTC* offers a peer-to-peer communication between peers, it still requires the exchange of signaling data like "*RTC offer and answer*" in order to establish the peer-to-peer communication channel. However, it is not specified in the *WebRTC* protocol how to exchange the signaling data. In our case, the

same channel used for launching applications can be used to exchange the *RTC signaling data* and the communication channel can be established. It is important to know that *WebRTC* also works across different networks even if both peers are behind NAT/Firewalls. Some network topologies are restricted and are not compatible with the *Session Traversal Utilities for NAT (STUN)* protocol used in *WebRTC*. In this case, the *Traversal Using Relays around NAT (TURN)* can be used to overcome this issue. This issue is not relevant if both application components are running on devices in the same local network.

- **Indirect communication**: this kind of communication requires a third entity that acts as a relay or proxy between the sender and receiver application components. The proxy must be well-known to the sender and receiver components. The sender application will get the endpoint of the proxy either after launching the receiver application or after joining an already running receiver application. As depicted in Figure 4.29 parts *b)* and *c)*, the sender and receiver components need to establish bi-directional communication channels to the proxy server which may run on a receiver device as in option *b)* or in the cloud as in option *c)*. In Web environments, *WebSocket* is one of the widely used protocols for duplex communication between client and server. In case *c)*, the sender and the receiver play the role of the clients, and the proxy plays the role of the server. The connection establishment process starts after the sender application launches or connects to the receiver application. During this step, the sender also sends a unique random token to the receiver and instructs it connect to the proxy server using the token. Similarly, the sender connects to the proxy using the same token. The proxy now puts both connections in the same pool and each message sent over one connection will be forwarded to the second connection and vice versa. It is also possible to add more than two connections in the same pool for example in a multiplayer game which allows sending a message to multiple receivers in the same pool. The total number of connections is the same as the number of senders and receivers.

After the communication channels between the application components of a multi-screen application are established, the three approaches *message-driven*, *event-driven* and *data-driven* can be implemented on top. As we mentioned before, the *message-driven* approach can be mapped directly to the underlying communication layer, and its implementation is straightforward.

Event-Driven

As described in Section 4.3.2, the *event-driven* approach requires an entity that acts as *Event Broker* and holds the event subscriptions. There are two ways to implement

this approach depending on which communication mechanism is used. In case of *indirect communication*, it makes sense to run the *Event Broker* on top of the *Communication Proxy*. Each sender or receiver can send the event subscription or publication data in *JSON* format to the proxy which forwards them to the *Event Broker*. The *Event Broker* holds a table which maps each connection of a multiscreen application component (either sender or receiver) to a list of event subscriptions. When an application component publishes data related to a specific event, it sends a publication message to the *Event Broker* which looks in the table for subscribers and notifies them by sending a *JSON* notification message. The following list shows the *JSON* structure of all message types exchanged between the application components and the event broker:

- ***Subscription***: {"*type*": "*subscribe*", "*event*": "*foo*"}
- ***Unsubscription***: {"*type*": "*unsubscribe*", "*event*": "*foo*"}
- ***Publication***: {"*type*": "*publish*", "*event*": "*foo*", "*payload*": "..."}
- ***Notification***: {"*type*": "*notify*", "*event*": "*foo*", "*payload*": "...", "*publisher*": "*pid*"}

The implementation of the *event-driven* approach in a decentralized environment follows the principles of the Gossip protocol [143] for spreading information across nodes in a peer-to-peer network. For this, each device runs an *event broker proxy* locally and offers the same interfaces and *JSON* messages as the *event broker* described above. The *event broker proxy* also consists of a table for event subscriptions but holds only local subscriptions. In this case, the publisher needs to send the event to all other peers of a multiscreen application. Since the publisher is not necessarily connected to any other peer, it sends the event to known peers first. The *event broker proxy* running on each of these peers will check if there are subscriptions for the received event and notify the subscriber components if needed. Furthermore, each *event broker proxy* will resend the event to known peers until it has been propagated to all peers. However, this solution still has a drawback since the propagation of the event will continue recursively in an endless loop. To overcome this issue, we extended the event propagation algorithm with a function that checks if the event was already received by a peer and in this case, the event will be dropped. The easiest option to implement the check function is to assign a unique random identifier to each event before publishing it for the first time. In addition to the subscription table, each *event broker proxy* needs a second table which holds received events. If the event is received for the first time which means that the event is not in the table, then it will be added, and the event will be sent to other peers. In case the event was already added to the table, it will be dropped and not sent to other peers. Since the number of events may increase rapidly, the size of the event table will also grow over time which requires more storage, and the lookup will take more time. A solution for this issue is to use an additional time-to-live attribute *ttl* either for each event or globally for all events. All events whose *ttl* has expired will be removed

from the table. Below is an example of the publication *JSON* message with a *ttl* of 5 seconds:

- ***Publication***: *{"type": "publish", "event": "foo", "payload": "...", "id": "e1", "ttl": 5}*

All other messages will remain the same as for the centralized approach.

Data-Driven

In this section, we will discuss the implementation of the *data-driven* approach introduced in Section 4.3.3. The implementation provides the following functions that allow applications to use this approach through corresponding APIs:

- **Initialization**: An application component creates a new object with a given name and optional initial state in *JSON* format using the `object()` method. If an object with the same name has already been created before (e.g. by another component of the same application), it will be retrieved, otherwise a new object will be created with the name and the initial state passed as input. In both cases, the application will be notified when the object is ready. For example, `object("foo", {"bar": 1}).then(callback)` creates a new object with the name `"foo"` and initial state `'bar': 1`. The callback function, e.g., `callback = function(foo){/* use foo*/}`, will pass the object `foo` to the application.
- **Read**: Once the object is ready, the application can access it as any *JSON* object. For example, `var x = foo.bar` can be used to read the value of the property `bar`.
- **Update**: Similar to the read operation, the application can also update the object as any *JSON* object. For example, *foo.bar = 2* sets the value of the property `bar`. The underlying synchronization protocol used in the implementation propagates the changes to other peers or clients. Since any manipulation on the object may result in an inconsistent state and rollbacks may be applied at any time, it is important to notify the application in order to react to this changes.
- **Notification**: Since the object can be manipulated by other components of a multiscreen application, it is important to notify the application about these updates. Therefore, the object should provide interfaces to allow the application to observe the value of any property in the object. For example, `foo.observe("bar", function(newVal, oldVal){ /* ... */})` notifies the application any time the value of `bar` is changed. The old and new values will be passed to the application in the listener function.

It is important to mention that at least one of the synchronization algorithms **Lock-step Synchronization** [126], **Bucket Synchronization** [129], **Time Warp Synchronization** [132] or **Trailing State Synchronization** [132] described in Section 4.3.3 must be implemented to keep the state of the object between the application components in sync. In case multiple synchronization algorithms are supported, the `object()` interface should to be extended to allow the selection of a specific algorithm. For example, `object("foo", {"bar": 1}, "trailing-state")` creates a new object and tells the underlying system to select the trailing state algorithm for synchronization. This way, application developers can select the algorithm that best suits their requirements. Other synchronization algorithms that are not listed in this thesis should be supported in a similar way without changing the interfaces for the application.

4.6.3 Application Runtime

Since our focus is on Web technologies, we extended existing browser-based runtimes such as *WebViews* [144] on mobile platforms (Android and iOS) and Chromium [145] on desktop with multiscreen APIs and used them as runtime environement for multiscreen application components. These APIs enable access to core multiscreen features provided by the underlying layer as defined in the UML diagram depicted in Figure 4.22. Web technologies were selected because of their cross-platform support especially on receiver devices like HbbTV, SmartTVs, and streaming devices. On some devices like Chromecast, this is the only type of supported technology. In addition to Web technologies, some platforms support also native applications like Android Apps in the case of Android TV and tvOS Apps in the case of Apple TV. Therefore, we will use the term User Agent (UA) in this section which cover all types of applications. A native application runtime is also considered as a UA which acts on behalf of the user and launches native applications instead of Web pages. There are efforts to unify the launch APIs of Web and native Apps by using *Uniform Resource Identifiers (URIs)* [146]. The most relevant part of a *URI* is the scheme [147] which can be used by the underlying platform to identify the corresponding user agent. for example, *URIs* with the scheme *http://* or *https://* are launched in a Web browser while *URIs* with the scheme *youtube://* are launched in the YouTube native App if it is available. We will differ between the following three implementations for the application runtime:

Multiple User Agents: This option considers a user agent for each device involved in the multiscreen application and responsible for the execution of the corresponding Composite Application Component assigned to that device. Figure 4.30 illustrates this implementation option with a multiscreen application assigned to two devices. The dotted red line between the *Multiscreen APIs* represents the communication

channel between the application components running on the two devices. We

Figure 4.30.: Multiple User Agents

implemented this option as proof-of-concept using the *DIAL* protocol [39] that launches a User Agent called *FAMIUM*. *FAMIUM* is an extended Web Browser that implements the *Multiscreen API* and runs a *DIAL Client* on sender devices and a *DIAL Server* on receiver devices. Furthermore, the *FAMIUM* receiver device runs a *WebSocket Server* as a communication proxy between the sender and receiver applications. The *WebSocket Server* can also be hosted anywhere in the cloud. The *FAMIUM* sender implementation is available for Android while the *FAMIUM* receiver implementation is available for all desktop platforms as *Node.js* module.
Furthermore, we provide a pure *JavaScript* implementation for this option which can be integrated into any Web application without the need to extend the Browser. This implementation includes a *JavaScript* client library and a server as *Node.js* module. The only limitation of this implementation is the discovery. It is not possible to discover other devices in the local network through a *JavaScript* API in the Browser due to security and privacy reasons. For example, a Web page could discover and connect to a network attached storage or get access to other network connected devices in the home and transfer data to a server without the user notice anything. Also, a Web page can use the metadata of discovered devices like serial number or unique device identifier to create a fingerprint and track all devices in the home visiting the same web page. Therefore, this implementation provides a fallback for discovery using a manual pairing of devices via PIN or QR code.

Single User Agent: This option considers a user agent on a device that executes multiple Composite Application Components each in a separate execution context. The Application Components are implemented in the same way as for multiple user agents since they use the same *Multiscreen API* which provides the same interfaces in all implementations. Figure 4.31 illustrates this option with a multiscreen application assigned to two devices. As we can see, the first device runs two Composite Application Components in the same user agent but in two different execution contexts. UA_1 executes the first Application Component assigned to device D_1 where the UI output is displayed on the same device D_1. The Application Component assigned to device D_2 is also executed on device D_1 in another execution context inside of UA'_1. However, since the Application Component is assigned to device D_2, the UI needs to

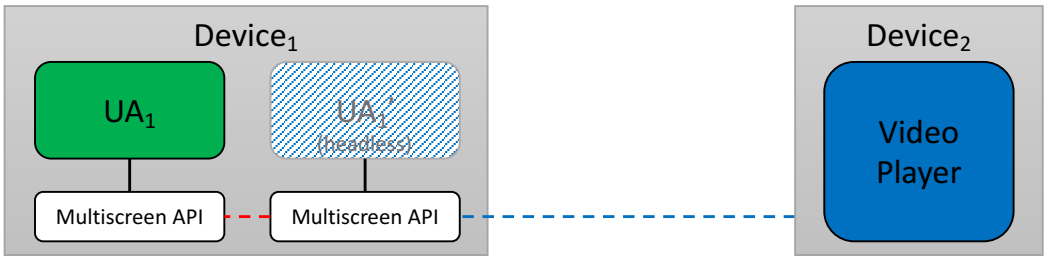


Figure 4.31.: Single User Agent

be also displayed on D_2. Therefore, this Application Component will be rendered in silent mode. This means that the UI will not be displayed on device D_1, but it will be captured as a video stream and sent to device D_2 which needs only to display the received video. As we can see, the dotted red line between the *Multiscreen APIs* of the two execution contexts represents the local communication channel between the two application components. On the other side, the dotted blue line represents the cross-device communication channel for sending the captured video stream to D_2. It is important to use suitable protocols designed for UI sharing such as *Airplay* and *Miracast*.
As proof of concept, we implemented this option for the *iOS* and *Android* platforms. The *iOS* implementation is based on the *Airplay* protocol [6] and supports *Airplay* receivers like *Apple TV* while the Android implementation is based on the *Miracast* protocol [8] and supports *Miracast* receivers like new Smart TV models.

Cloud User Agent: This option considers a user agent running in the cloud and executes multiple Composite Application Components in silent mode and each in a separate execution environment. The UI of each component will be captured and sent to the corresponding device as a video stream. Furthermore, all user inputs like keyboard and touch screen are captured and sent to the cloud user agent which triggers the input events in the corresponding execution context. Figure 4.32 illustrates this option with a multiscreen application assigned to two devices. We can also see in this example similar to the Single User Agent case, that the communication between the *Multiscreen APIs* is local (dotted red line). We can also see that the video stream of the captured UI of both application components will be sent over the internet to the corresponding devices (dotted green and blue lines). We implemented this option as proof-of-concept using the *Chrome Embedded Framework (CEF)* [145] as a user agent for the silent rendering of the application components. We also experimented with the following profiles for capturing and streaming of the rendered UI:

- Capturing and encoding the UI output as images in *Bitmap*, *PNG* and *JPEG* formats and sending them to the client over *HTTP* or *WebSockets*.

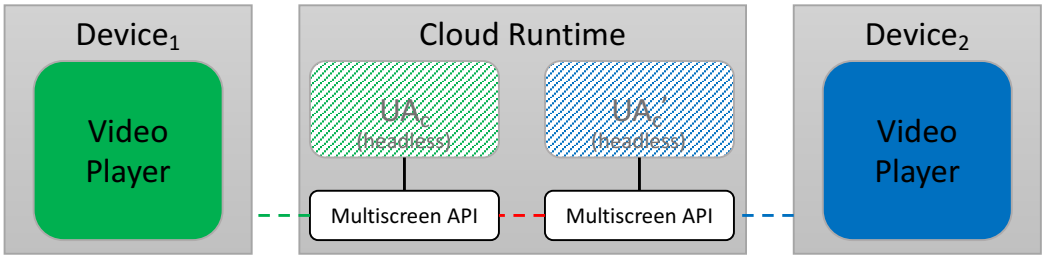


Figure 4.32.: Cloud User Agent

- Capturing and encoding the UI output as video stream using the codecs *h264* and *VP8* and sending it to the client over *HTTP* or *WebSockets*.
- Capturing and sending the UI output using *WebRTC* media streams.

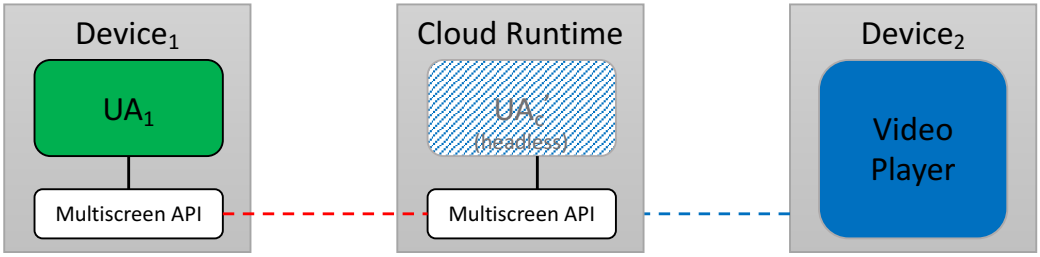


Figure 4.33.: Combination of Multiple and Cloud User Agents

Even though we discussed the three implementation options for the Application Runtime in this section, a combination of these implementations is also possible. Figure 4.33 shows an example that combines the multiple user agent implementations with the cloud user agent implementation. In real-world scenarios, it makes sense to use this option for low capability receiver devices like Set-Top-Boxes that are not capable of running the receiver application. The sender application will be executed and rendered on the sender device, e.g., a smartphone. In Section 6.1 we will provide an evaluation of the implementation options we considered in this section and discuss the advantages and disadvantages of each implementation.

5 Multimedia Streaming in a Multiscreen Environment

In the previous chapter, we focused on multimedia applications in a multiscreen environment, introduced a multiscreen application model based on Web technologies and discussed all potential implementation options. In this chapter, we will focus on multimedia content in a multiscreen environment. After we having introduced state-of-the-art technologies for multimedia content preparation, streaming and playback in Chapter 2, we will now discuss the applicability of these technologies in multimedia applications. in general, multimedia content refers to video, audio, and image but our focus will be on video which is the most important and challenging format. Besides regular fixed-perspective videos, we will investigate immersive videos especially 360° videos. This new video format is highly challenging regarding different aspects like content production, preparation, storage, delivery, and playback especially in a multiscreen environment where devices have different characteristics like processing capabilities, supported video codecs and connectivity. This chapter is structured as follows: Section 5.1 investigates the different methods for sharing multimedia content on different devices while Section 5.2 focuses on spatial multimedia content. Section 5.2.2 deals with the synchronization of multiple media streams on multiple screens. Afterward, Section 5.3 discusses the different approaches for immersive media playback and provides an innovative solution that enables 360° video playback on a wide range of devices especially on TVs. Finally, Section 5.3.6 gives an overview of our implementation of the most relevant components.

5.1 Multimedia Sharing and Remote Playback

Multimedia sharing is one of the most important and widely deployed multimedia application scenarios. The basic flow is depicted in Section 3.1.1. This scenario can be realized as a multiscreen application with sender and receiver components following the models and concepts we introduced in the previous chapter. The receiver is just a simple Web component that includes an HTML video element in fullscreen, launched remotely by the sender application. To control the playback of the video on the receiver device, one of the three approaches introduced in Section 4.3 can be applied. The event-driven approach fits well for this kind of application scenarios and can be implemented easily. The sender application shows the video

controls such as Play/Pause buttons and progress bar showing the current video time and triggers multiscreen events each time the user makes an interaction. On the other side, the receiver application component plays the video and listens to multiscreen events triggered on the sender. It also triggers multiscreen events each time the playback state changes which can be used on the sender to update the video controls. Listings 5.1 and 5.2 shows parts of the sender (*AACControl*) and receiver (*AACPlayer*) atomic application components for this scenario.

```
/* Video Control AAC */
class AACControl extends AAC {
 connectedCallback() {
  ...
  var aac = this;
  playBtn.onclick = function(){
   aac.publish("playClick");
  };
  aac.subscribe("timeUpdate",t=>
    {
   progress.value = t;
  });
 }
}
```

Listing 5.1: Multimedia Sharing Sender

```
/* Video Player AAC */
class AACPlayer extends AAC {
 connectedCallback() {
  ...
  var aac = this;
  vid.ontimeupdate = function(){
   var t = vid.currentTime;
   aac.publish("timeUpdate", t);
  };
  aac.subscribe("playClick",e=>{
   vid.play();
  });
 }
}
```

Listing 5.2: Multimedia Sharing Receiver

As we can see, the implementation of the multimedia sharing scenario is straightforward, but since it is a common scenario, it makes more sense to extend the multiscreen model to a new *Remote Playback API* that enables multimedia sharing using a simple and easy to use interface. In this case, the developer only needs to implement the sender application which uses the new API to play the media remotely on the receiver device. The advantage of this approach is that it is not only easier to implement, but also supports a wider range of receiver devices that provide media rendering capabilities but not necessarily an application runtime with a complete stack (to run receiver applications). The *Remote Playback API* adds a new method $setMedia(media)$ to the $Device$ class depicted in the Multiscreen UML diagram in Section 4.5.2. $setMedia$ can be called on a discovered device instance and accepts as input an HTML media element ($HTMLVideoElement$ or $HTMLAudioElement$). In this case, the media playback will be stopped on the sender device and continues on the receiver device. If the input passed to this method is $null$, then the media playback will be stopped on the receiver device and continued on the sender device. The sender application only needs to operate on the HTML media element passed as input parameter to control the playback on the receiver device, subscribe to player events or read playback info like the current playback time.

```
<button id="cast">Cast</button>
<video id="video">
 <source src="video.m3u8" type="application/x-mpegURL">
 <source src="video.mpd" type="application/dash+xml">
```

```
 <source src="video.mp4" type="video/mp4">
 <source src="video.webm" type="video/webm">
</video>
<script>
 var video = document.querySelector("#video");
 var castBtn = document.querySelector("#cast");
 var device;
 var msa = this.msa;
 msa.ondevicefound = function(e){
  msa.stopDiscovery();
  device = e.device;
 };
 castBtn.onclick = function(){
  device && device.setMedia(video);
 };
 msa.startDiscovery({canPlay: video});
</script>
```

Listing 5.3: Multimedia Sharing Receiver

Listing 5.3 shows an example for multimedia sharing using the new API. As we can see, the $startDiscovery$ function uses the video element as a filter ($canPlay$) which is necessary to find only devices that can play the requested video. The HTML video specification enables providing multiple sources for the same video, e.g., for different streaming formats and video codecs. The browser selects the best suitable source it supports. This can also be used during discovery to find only devices that can play at least one of the available sources. The video element in the example above provides four sources: the first two are adaptive streaming formats HLS [68] and DASH [67] while the last two are regular single file *MP4* and *WebM* videos. Since the landscape of media formats and codecs is highly fragmented, it is essential for content providers to know the devices and platforms that need to be supported and provide compatible video sources. Another method is to support widely adopted container formats such as MP4 and video codecs such as H.264 as a fallback to adaptive streaming formats like DASH and HLS which are the preferred formats in a multiscreen environment. As the name "adaptive streaming" suggests, the main reason why it fits best for multiscreen applications is that the video playback adapts automatically to the device capabilities and network conditions. When the user starts the video on a small screen like a smartphone, then the adaptation set that corresponds to the screen resolution and available bandwidth will be selected automatically in the player. If the user starts the remote playback on a UHD TV, then the UHD adaptation set will be selected if it is available and the network condition supports this selection.

5.2 Spatial Media Rendering for Multiscreen

Spatial Media Rendering is an approach for playing a spatial sub-part of a video on a target display. The visible area of the video is called Region-of-Interest (ROI) which is defined as a rectangle *(x,y,w,h)* where *(x,y)* is the coordinate of the top left corner and (w,h) is the dimension of the viewport. The selection of supported ROIs varies from use case to use case. The example depicted in figure 5.1 shows three versions of a video with three different resolutions low (a), medium (b), and high (c). The dimension of the viewport is, in general, the same as the dimension of the target display. If the viewer wants to see the entire video, then the low-resolution version is selected. On the other hand, if the user wants to display a sub-part of the video (which is relevant for videos recorded using wide view cameras) while retaining the output resolution and quality, then a higher resolution version needs to be selected. In this case, the user can zoom into the video without affecting the output quality. Since the transition between two levels (for example from level *a)* to level *b)* depicted in Figure 5.1) takes some time until the video segments of the new level are streamed to the client, the player can zoom in the video of the source level which results in a lower quality for the selected ROI during the transition time. The transition time depends on several factors like latency, bandwidth and segment duration. A side effect of spatial media rendering is the wasted bandwidth since

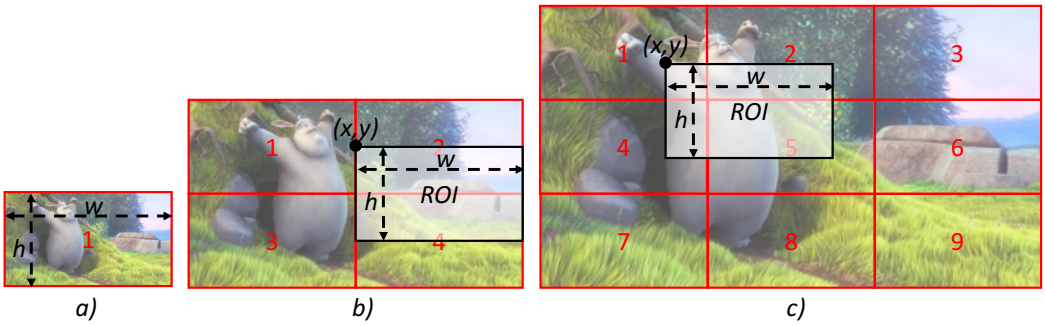


Figure 5.1.: Spatial Media

more video data is streamed to the client than the amount of video data actually displayed to the user. In the example depicted in Figure 5.1, the ROI in level b) is 25% $(1/4)$ of the entire video and in level c) about 11% $(1/9)$. This issue is already addressed in current research, for example, [104] and [107] we discussed in Section 2.4.2 use state-of-the-art video codecs that support tiling like *HEVC* [148]. In this case, the video will be split into multiple tiles that can be requested individually by the client. Furthermore, each tile can be provided in multiple bitrates which allows the client to request tiles that intersect with the ROI in a higher bitrate than the tiles outside of the ROI. For example, the ROI in video *b)* intersects with tiles *2* and *4* and in video *c)* with tiles *1*, *2*, *4* and *5* and only these tiles need to be streamed in a higher bitrate.

Until now, we considered the rendering of partial video content on a single screen, but what if we want to show the whole video on a multi-display installation in a certain arrangement, e.g., video wall which consists of a *NxM* matrix of single displays. The total resolution of the whole video wall is, in this case, *NxMxWxH* where *WxH* is the resolution of a single display. For example, a *2x2* video wall composed of 4 Full HD (1920x1080) displays has a total resolution of (2x1920x2x1080) or (3840x2160) which is equal to 4K. The following sections describe the different methods for content preparation and synchronized playback for the video wall scenario which may also be applied to other application use cases where synchronization between multiple media streams is required.

5.2.1 Content Preparation

Each display of the video wall should play the ROI that corresponds to its position in the matrix. One option to achieve this is depicted in Figure 5.2 a). It streams the whole video to each display which selects and shows only the ROI that corresponds to its position in the matrix. The disadvantage of this method is the wasted bandwidth since the same video content is sent to the displays multiple times. The option

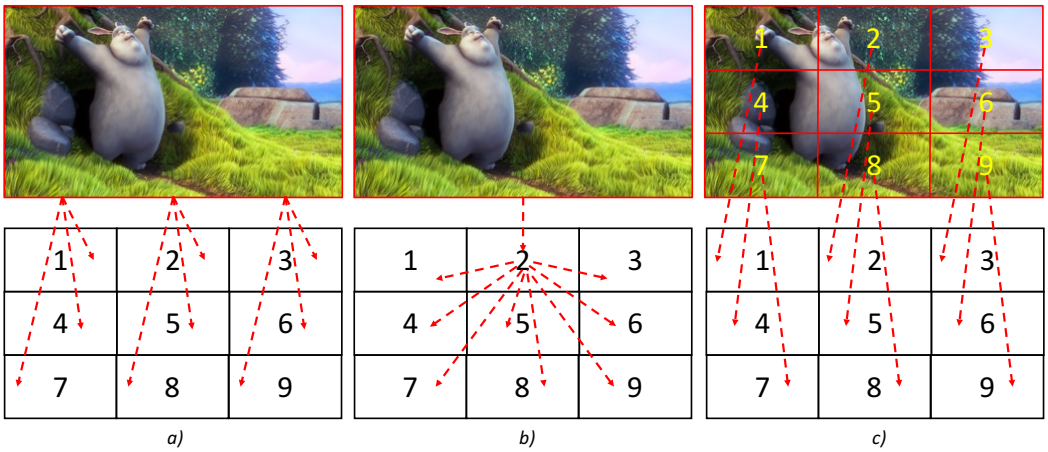


Figure 5.2.: Video Wall

depicted in 5.2 b) addresses this issue and streams the video only to one master display which distributes it to all other displays. This option solves the bandwidth problem, but it still has the disadvantage that the player on each display needs to decode a high-resolution video even though it displays only a small part to the user. Furthermore, this option requires a local peer-to-peer communication mechanism among the displays which is not always guaranteed. The last option depicted in Figure 5.2 c) is the recommended one and uses the same technique for spatial media rendering as described in Section 5.2. The video is split into several tiles in the same order as the displays in the video wall, and each display requests the tile that corresponds to its position. The only difference compared to the spatial media

rendering method is that there is no need to use special video codecs that support tiling like HEVC since each display plays only one single tile. Therefore, there is no need for merging tiles, and any video codec like H.264 can be used.

5.2.2 Seamless, Consistent and Synchronized Playback

As in most media-related scenarios, adaptive streaming and optimal usage of available resources are essential for enabling seamless video playback. This also applies to the video wall scenario, but it raises additional challenges which are not necessarily relevant for video playback on a single screen. We will address these new challenges by considering option c) described above which splits the video into multiple tiles that are played individually on the corresponding displays of the video wall. To enable adaptive streaming on each display, the video tiles should be provided in different bitrates. It is important that all displays play the tiles with the same bitrate at any time to avoid inconsistencies in the video quality between the displays. Furthermore, the players on the different displays should playback the tiles with frame-accurate synchronization. To achieve this, the players on all displays should coordinate among each other to control the buffering behavior and playback state by exchanging relevant playback metrics like player state and time, available bandwidth on each display and the amount of buffered video data. We assume that the players can communicate with each other using one of the multiscreen approaches introduced in the previous chapter. The synchronization algorithm is described below. It follows a master/slave approach, where one of the displays, i.e., the first display in the video wall, is the master (also called coordinator) and all other displays are the slaves.

The master part is depicted in Algorithm 1 while the slave part is depicted in Algorithm 2. Furthermore, Figure 5.3 highlights the most important steps of the algorithm in a UML sequence diagram. As we can see, the master algorithm keeps the amount of buffered video content on each display at the same level by synchronizing the HTTP requests to load the segments. A display can request a tile segment k only if all displays already buffered tile segment $k-1$. Furthermore, the master determines the bitrate level before sending each request by considering the smallest bandwidth available on the displays. Since the bandwidth may vary over time, it is important to measure it after each HTTP request and update the bitrate level accordingly. At the beginning, the lowest bitrate level is selected. To synchronize the playback across all displays, the master periodically sends its playback time along with the system time to all slave displays. The slave part of the algorithm handles events from the master to load and buffer tile segments and to synchronize the playback state and time with the master. The slave receives periodically the master video time and master system time which are used together with the slave video time and slave system time to calculate the time difference between the master

Algorithm 1 Master Algorithm

Input: I ▷ Number of displays
Input: J ▷ Number of bitrate levels
Input: K ▷ Number of video segments
Input: $bitrate_j$ ▷ Bitrate of level *j*
Input: $tile_{ijk}$ ▷ Tile segment *k* for display *i* and with bitrate level *j*
Define: $display_i$ ▷ Display with index i. $Display_1$ is the master
Define: $j \leftarrow 1$ ▷ current bitrate level
Define: $k \leftarrow 1$ ▷ index of the current segment
Define: $player$ ▷ Video player on the master

1: **function** INITIALIZE
2: **send event** READY **to** $display_1$
3: **end function**
4: **upon receiving event** READY **from** $display_i$ **do**
5: $display_i.state \leftarrow ready$
6: **if** $\forall\ i = 1..I,\ display_i.state = ready$ **then**
7: BUFFERNEXTSEGMENT( )
8: **end if**
9: **end event**
10: **upon receiving event** BUFFERED($bandwidth$) **from** $display_i$ **do**
11: $display_i.state \leftarrow buffered$
12: $display_i.bandwidth \leftarrow bandwidth$
13: **if** $\forall\ i = 1..I,\ display_i.state = buffered$ **then**
14: BUFFERNEXTSEGMENT( )
15: **end if**
16: **end event**
17: **upon receiving event** REQUEST($tile$) **from** $display_1$ **do**
18: $req \leftarrow$ CREATEHTTPGETREQ($tile$)
19: $res \leftarrow$ SENDREQANDWAITFORRES(req)
20: APPENDTOVIDEOBUFFER($player, res.data$)
21: **send event** BUFFERED($res.bandwidth$) **to** $display_1$
22: **end event**
23: **upon receiving event** TIMEUPDATE($videoTime$) **from** $player$ **do**
24: $systemTime \leftarrow$GETSYSTEMTIME( )
25: **for all** $i = 2..I$ **do**
26: **send event** TIMEUPDATE($videoTime$, $systemTime$) **to** $display_i$
27: **end for**
28: **end event**
29: **function** BUFFERNEXTSEGMENT
30: **if** $k \leq K$ **then** ▷ End of video not reached yet
31: $j \leftarrow$ DETERMINEBITRALELEVEL()
32: **for all** $i = 1..I$ **do**
33: $display_i.state \leftarrow ready$
34: **send event** REQUEST($tile_{ijk}$) **to** $display_i$
35: **end for**
36: $k \leftarrow k + 1$
37: **end if**
38: **end function**
39: **function** DETERMINEBITRALELEVEL
40: $bandwidth \leftarrow min(display_{i=1..I}.bandwidth)$
41: $level \leftarrow l$ where $bitrate_l < bandwidth \leq bitrate_{l+1}$
42: **return** $level$
43: **end function**

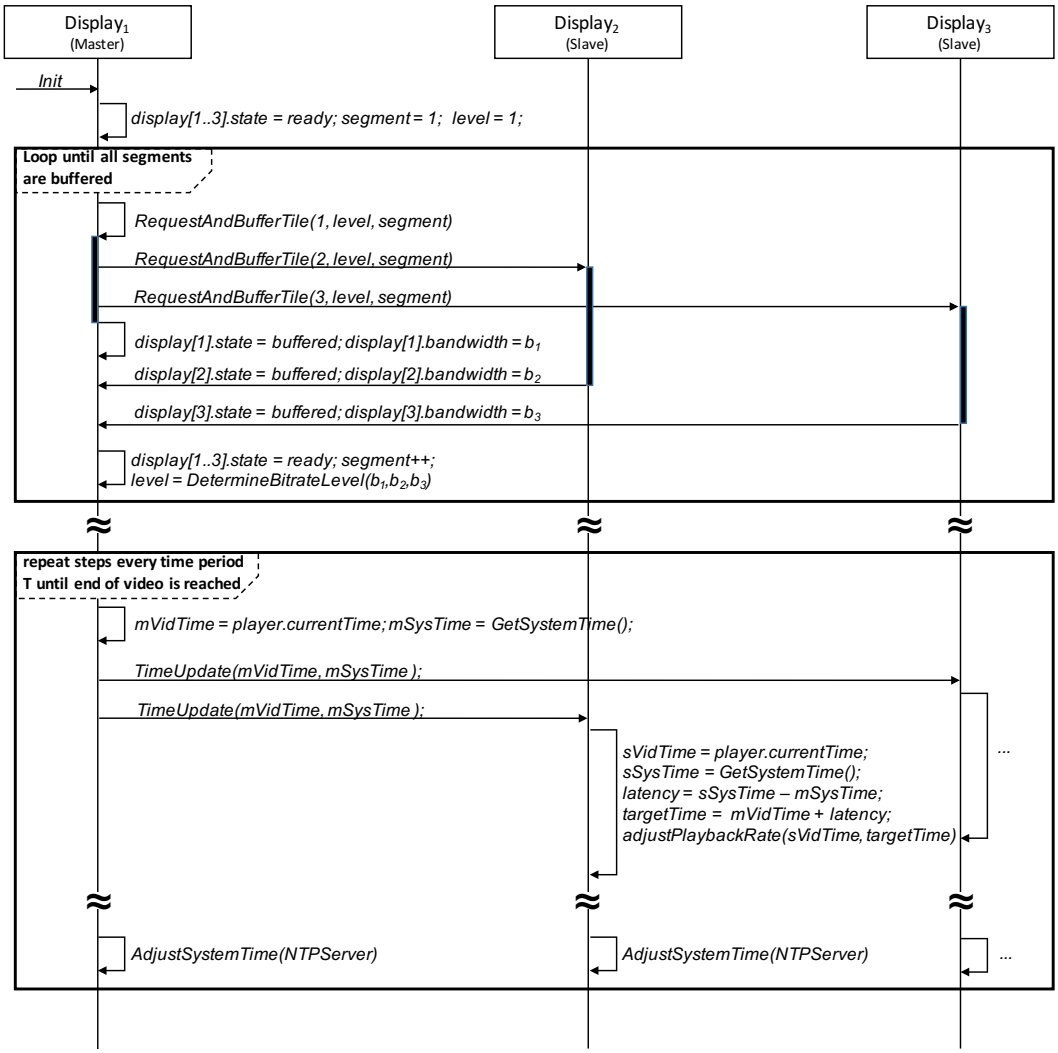


Figure 5.3.: Video Wall Synchronization Algorithm Sequence Diagram

and slave players. If the time difference exceeds the threshold of a given accuracy which is, in general, the time of a single frame ($40ms$ for video frame rate of $25fps$), then the slave needs to update its player time accordingly. There are two methods to do this: *a)* seek immediately to the newly calculated target video time or *b)* change the video playback rate to a value so that the target video time can be achieved after a specific time T. In the slave algorithm, method *b)* is selected since video seeking is not accurate compared to changing the playback rate in most player implementations. Playback rate of 1 means that the video plays at normal speed, while values < 1 or > 1 indicate that the video playback speed is slower or faster than the normal speed. The playback rate is updated after each *timeupdate* event until the difference between the master and slave video times stabilizes below the accuracy threshold. In some situations, the video playback rate may not stabilize quickly and cause the player to oscillate. This can happen if the underlying player implementation does not support accurate setting of the playback rate. Some old players even support only few playback rate factors, e.g., 0.5 or 2. A way to avoid the

oscillating behavior is to trigger the *timeupdate* event more often and by increasing the accuracy threshold. In the evaluation section we will show that we can achieve nearly frame-accurate synchronization while avoiding oscillation in the video playback. The calculation of the video playback rate on the slave display after each

Algorithm 2 Slave Algorithm

Input: $accuracy$ ▷ max allowed diff between master and slave video times
Input: $T > accuracy$ ▷ max time needed to achieve the playback position of the master
Define: $master$ ▷ master display
Define: $player$ ▷ Video player on slave

1: **function** INITIALIZE
2: **send event** READY **to** $master$
3: **end function**

4: **upon receiving event** REQUEST($tile$) **from** $master$ **do**
5: $req \leftarrow$ CREATEHTTPGETREQ($tile$)
6: $res \leftarrow$ SENDREQANDWAITFORRES(req)
7: APPENDTOVIDEOBUFFER($player, res.data$)
8: **send event** BUFFERED($res.bandwidth$) **to** $master$
9: **end event**

10: **upon receiving event** TIMEUPDATE($mVidTime, mSysTime$) **from** $master$ **do**
11: $sSysTime \leftarrow$GETSYSTEMTIME( )
12: $sVidTime \leftarrow player.time$
13: $latency \leftarrow sSysTime - mSysTime$
14: $targetTime \leftarrow mVidTime + latency$
15: **if** $|sVideoTime - targetTime| \leq accuracy$ **then**
16: $player.playbackRate \leftarrow 1$
17: **else**
18: $player.playbackRate \leftarrow 1 + \frac{targetTime - sVideoTime}{T}$
19: **end if**
20: **end event**

received *timeupdate* event is highlighted in Figure 5.4. It is important to mention that it is not a method for clock sychronization. The video wall synchronization algorithm above expects that the master and slave system times ($mSysTime$ and $sSysTime$) are synchronized with a time server. Clock synchronization is a well researched topic and there are already available solutions for it. Probabilistic clock synchronization [149] is one of the simplest algorithms for clock synchronization in distributed systems. A master device can take the role of a time server which handles requests from other computers (slaves) in the network by responding with its local time.The Round-Trip-Time (RTT) for the message exchange between the master and the slaves is taken into account for calculating the time on the slaves. Another solution is the Network Time Protocol (NTP) [150], an IETF standard for clock synchronization of computers on the Internet. It is based on a client-server model with

a network of time servers distributed around the globe. It can be also operated in local networks with a time server running on dedicated computer. NTP uses UDP to exchange messages between computers and time servers by taking network latency into account. In the implementation of the video wall synchronization, we used NTP for clock synchronization. Back to the method for calculating the playback rate on slave players, the latency between sending the *timeupdate* event on the master and receiving it on the slave is calculated as $sSysTime - mSysTime$. It means that at the time of receiving the event on the salve, the master video position was $targetTime = mVidTime + latency$ where $mVidTime$ is the master video position at the time of sending the event. If the difference between the slave video position ($sVidTime$) at the time of receiving the event and the calculated target video time ($targetTime$) is above the accuracy threshold, then the playback rate of the slave video will be set to a value r with the goal, that after a time T the master and slave video times are equal. In practice, it is nearly impossible to achieve the same values for the master and slave player times. This is the reason why the accuracy threshold was introduced. The algorithm can be used not only to synchronize video tiles on

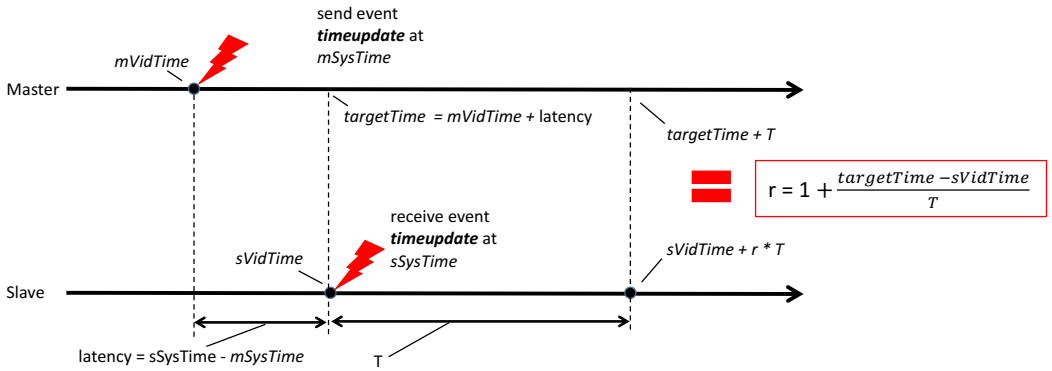


Figure 5.4.: Calculation of slave video playback rate r

multiple displays but also for any multi-stream synchronization on multiple devices like the scenario described in Section 3.1.3. In this case, a video stream playing on TV will be synchronized with one or more audio streams on mobile devices. Another application domain that requires synchronization of multiple videos is multi-view and multi-camera streaming, especially for sports events. For example, during live streaming of a car race, the viewer can follow the main view on the TV and select an alternative camera stream on the companion device, i.e., the camera installed in the car of the favorite driver.

5.3 360° Video for Multiscreen

In the previous sections, we focused on media sharing, remote playback, synchronization of multiple media streams, and spatial media in a multiscreen environment.

In the last section of this chapter, we will investigate immersive media formats especially 360° videos on various playback devices like Head-Mounted Displays (HMDs), Smartphones and TVs. We will provide a solution that enables the application scenario described in Section 3.1.6. The work introduced in this section is based on the four accepted publications [19], [151], [20] and [21] submitted by the author of this thesis at international conferences.

5.3.1 Challenges of 360° Video Streaming

Immersive video has been around for some time, dating back to the "*A Tour of the West*" short movie from 1955 [152] using Disney's "*Circle-Vision 360°*" technology [153]. It re-emerged a couple of times, though mostly as a showcase exhibit at trade fairs rather than as a real-world video format in its own right. Only in recent years the market situation changed. Affordable cameras with sufficient resolution became available to allow professionals and interested amateurs to create 360° movies. Also, stitching software became good enough to stitch the videos and hide the seams with reasonable quality. Networks became fast enough to allow end-users to stream 360° video content in reasonable quality. Smartphones and tablets are sufficiently powerful and have the necessary sensors to handle the content up to certain video quality and now can react to view changes without noticeable delay. As a result, major video and social media platforms such as YouTube and Facebook have introduced 360° video on a variety of mobile devices and head-mounted displays. Also, some providers enabled 360° video on TV devices like YouTube which supports 360° on new high-end Android TVs and Facebook which supports 360° on Apple TV. Although 360° video technology has improved over time in the last years, it still faces many challenges and limitations which significantly limit the immersive experience for the user. In the following, we will discuss these challenges in more detail:

Bandwidth: Current 360° video streaming solutions use the same streaming technologies and content delivery networks as for regular videos. On the distribution side, almost all current solutions stream the full 360° content to the end-user device, whereby only a small area of the sphere is presented to the viewer while the other parts are disregarded, causing a huge bandwidth consumption. While the actual amount of video content displayed to the user depends from multiple factors like video projection (see Section 2.3.4), supported codecs (see Section 2.3.4), the viewing angle and the visible area, the viewer sees about 1/10 of the available sphere in average. Let us consider as an example the equirectangular video frame depicted in Figure 5.5. Two Field of Views (FOVs) with a horizontal angle of 90° (green area) and 120° (yellow area) are calculated from the equirectangular frame. The output FOV frames are depicted in Figures 5.6a and 5.6b. As we can see, the green part of the equirectangular video which is required to calculate the FOV with a

horizontal angle of 90° is about 1/12 of the whole video. In other words, 91.67% of the equirectangular video is streamed to the client but remains unseen. The same applies to the yellow part of the equirectangular video which is about 1/6 of the equirectangular video and needed to calculate the FOV with a horizontal angle of 120°. As we can see, the distortion in the calculated FOV becomes more visible for a wider angle of view. This is why most players consider a FOV with a horizontal angle of view between 90° and 100°. If the equirectangular video has a 4K resolution of

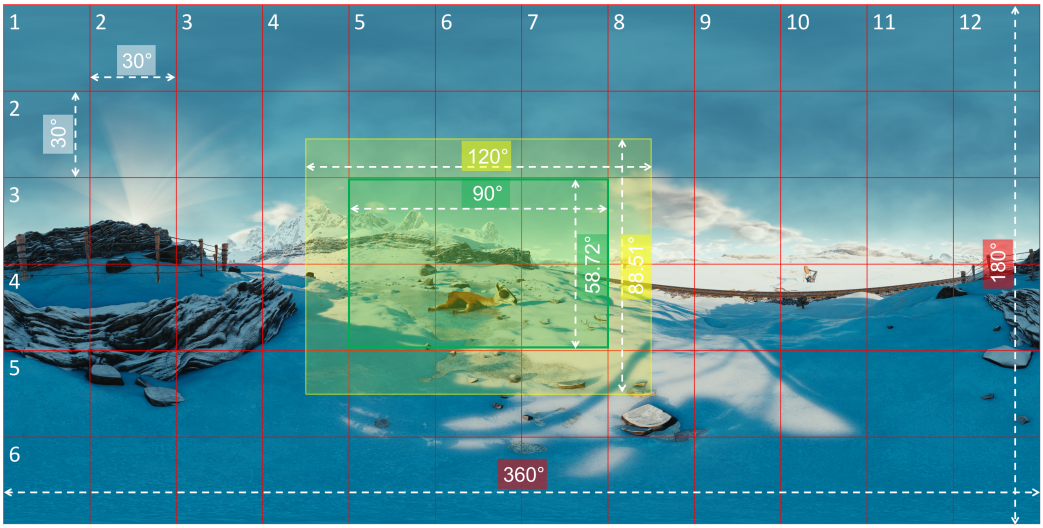
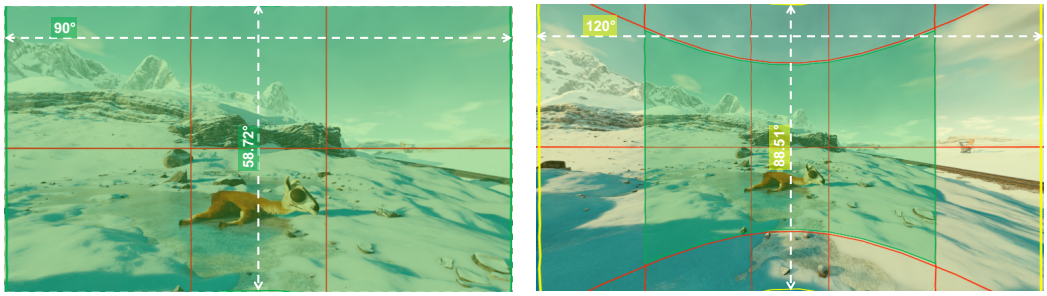


Figure 5.5.: Equirectangular 360° Video Frame



(a) FOV with a horizontal angle of 90° **(b)** FOV with a horizontal angle of 120°

Figure 5.6.: Calculated FOVs with two settings

4096x2048 pixels, then the approximate resolution of a FOV with 90° horizontal view angle is about 1024x576 (1024 is a quarter of 4096 since a 90° FOV is a quarter of the total 360°. 576 is used to get a 16:9 aspect ratio for the FOV) which is between SD (640x360) and HD (1280x720) resolutions. Inversely, to allow the end user to experience 360° content in 4K FOV resolution, i.e., on 4K TVs, the source 360° video must have a resolution of 16K (16384x8192).

Processing: In order to display a FOV of a 360° video to the user, the following three steps are required: 1) decode the 360° video (mostly equirectangular video),

2) calculate the FOV frames from the decoded 360° video frames by performing the geometrical transformation that corresponds to the used projection, and 3) render the calculated FOV frames. The decoding of the source 360° video requires more graphical processing resources compared to decoding regular videos with the same FOV resolution. This means that a device must be able to decode a 4K video in order to display an HD FOV or a 16K 360° video in order to display a 4K FOV. In addition to the encoding, the device must be able to perform the geometrical transformation of the FOV video in real-time. For example, the time limit to calculate a FOV frame from a 360° video frame with 30fps is 33.33ms (the display time of a single frame). Otherwise, the player will drop frames, and this will have a negative impact on the user experience. The processing costs for calculating a FOV frame

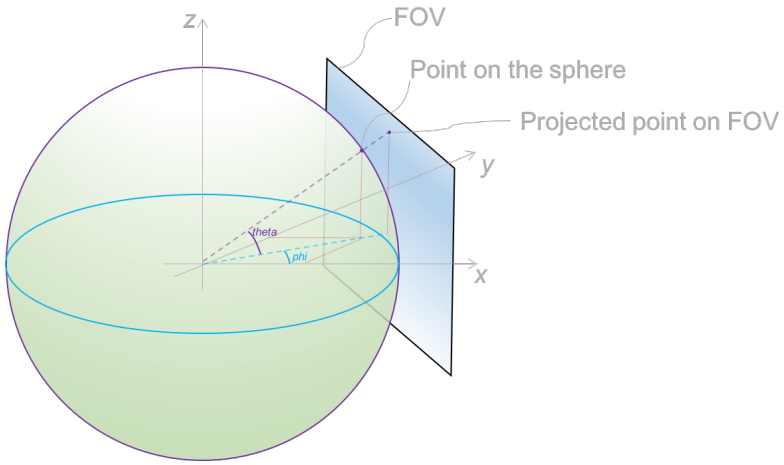


Figure 5.7.: Projection on FOV plane

increase proportionally to the resolution of the FOV. Figure 5.7 shows the projection of one point from the sphere on the FOV plane which represents a single pixel and therefore, the total number of projected points is equal to the total number of pixels in the FOV.

Video Encoding: In Section 2.3.4 we discussed state-of-the-art video codecs which can also be used for 360° videos. H.264 is one of the most supported video codecs compared to other codecs like HEVC and VP9. Hardware-accelerated decoding of H.264 videos is supported on almost any device. On the other hand, the codecs HEVC and VP9 provide better compression rates compared to H.264 according to [154]. Although these codecs provide better compression rates, many content providers still use H.264 in order to reach users on all devices and platforms. Furthermore, all evaluations of state-of-the-art video codecs consider mainly regular videos and not 360° videos which are more relevant for this work. In order to get more accurate results of codec compression efficiency for 360° videos, we evaluated multiple 360° YouTube videos which are available in different resolutions and with the codecs H.264 and VP9 (HEVC is not supported). According to YouTube's recommended

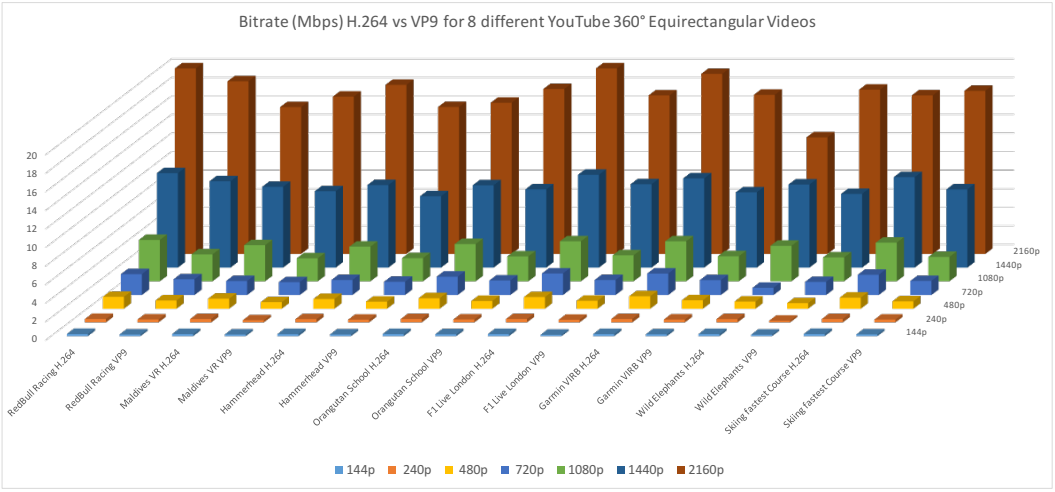
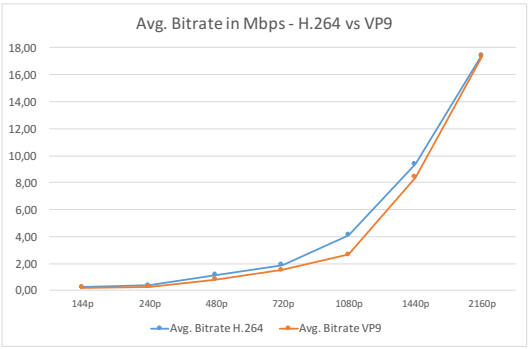


Figure 5.8.: Bitrates of 8 360° YouTube videos with varying output resolutions and codecs

upload encoding settings [155], the bitrate for a 4K H.264 encoded video with 30fps is about 45 Mbps while the recommended bitrate for an HD video also encoded with H.264 is about 5Mbps. In other words, an uploaded 360° 4K H.264 video has a 9 times higher bitrate than a FOV H.264 HD video calculated from it. After uploading a 360° video to YouTube, it will be processed, and multiple versions with different resolutions and codecs (H.264 and VP9) will be generated from the originally uploaded video. For our evaluation, we selected eight different 360° videos with a maximal resolution of 4K and measured the bitrate for each combination of output resolution and codec. The results are depicted in the chart in Figure 5.8. The average bitrates of all videos for the H.264 and VP9 codecs and with different resolutions is depicted in Figure 5.9. We see that the bitrate saving of



Res.	H.264	VP9	Saving
144p	0,24	0,19	19,44%
240p	0,37	0,29	22,62%
480p	1,15	0,80	30,09%
720p	1,87	1,53	18,18%
1080p	4,10	2,66	34,97%
1440p	9,38	8,40	10,47%
2160p	17,45	17,34	0,65%
FOV-%	10,72%	8,82%	

Figure 5.9 & Table 5.1: Avg. Bitrates in Mbps for codecs H.264 and VP9

the VP9 codec compared to H.264 is between 10% and 35% at resolutions up to 1440p (2K). This result is expected according to scientific publications that evaluate video compression standards [156] [157]. If it comes to 2160p (4K) resolution which is the minimum preferred resolution for 360° videos in order to provide a FOV resolution of nearly 720p (HD), the bitrates for H.264 and VP9 are nearly the same. This result shows that the compression efficiency of conventional codecs

such as H.264 and VP9 behaves differently between 2D and 360° equirectangular (EQR) video content. "*The main drawback of EQR is its latitude dependent sampling density unlike conventional 2D content*" [158]. This could be the reason why YouTube supports H.264 for 360° videos up to 4K resolution, while the maximum supported resolution of regular 2D videos using H.264 is 1440p (2K) and higher resolutions are only supported using the VP9 codec. H.264 is the better choice in case the compression efficiency is the same compared to other video codecs since H.264 with hardware-accelerated decoding is supported on nearly any device and platform with video playback capability. Let us now consider the amount of wasted bandwidth of 360° streaming using both codecs H.264 and VP9 . From Figure 5.9, we can see that the percentage of bitrate saving of HD FOV videos compared to 4K equirectangular videos is about 10,72% for H.264 and 8,82% for the VP9 codec. This means that on average around 90% of the bandwidth is wasted for streaming unseen content. In the next sections, we will introduce a solution for 360° video streaming and playback that addresses the main challenges discussed in this section.

5.3.2 Classification of 360° Streaming Solutions

A 360° video playout consists mainly of four key components as shown in Figure 5.10 (which does not include the content generation process). We assume that either a recorded 360° video or a 360° stream coming from a live source like a 360° camera is already available. The four components are:

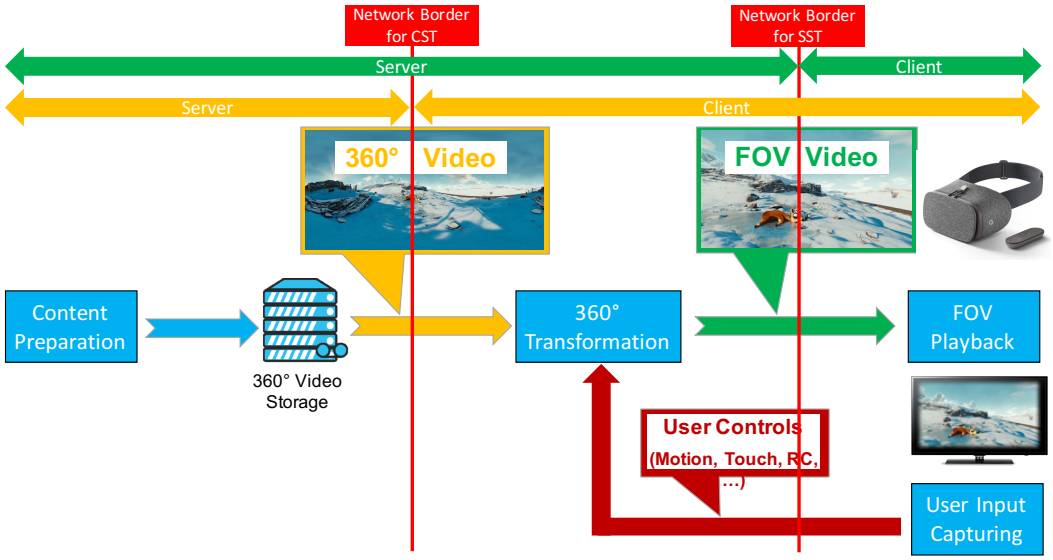


Figure 5.10.: 360° Playout - CST vs SST

- **Content Preparation**: This component includes all steps necessary to make the 360° video ready for streaming and playback on a set of devices and plat-

forms. In most cases, this component takes care of the generation of adaptive bitrate content, such as HLS and DASH. Also, this component may convert the projection used in the source 360° video to another (more efficient) projection. For example, most cameras produce 360° videos using equirectangular projection, but after converting a video to cube map projection, the video bitrate can be reduced by around 25%.

- **360° Transformation**: An important part of 360° solutions is the transformation of a projected 360° video to a FOV, which is essentially the 2D viewport of the user. This component takes the 360° content from the previous step and performs the geometrical transformation that corresponds to the used projection in order to calculate the FOV. It expects as input the FOV dimension (w, h, fov) and the center angle $(phi, theta)$. w and h are the width and height of the viewport in pixels, and fov is the vertical opening angle (the horizontal opening angle can be calculated from w, h, and fov).
- **FOV Playback**: This component is just a video player that renders the FOV calculated in the previous step on various types of end-user devices like HMDs, TVs, and mobile devices.
- **User Input Capturing**: This component captures the user inputs control the view port. The captured inputs are used to calculate the center angle $(phi, theta)$ of the FOV before sending it to the *360° Transformation* component. The captured user inputs can vary from device to device. For example, motion sensors are used on HMDs to detect head movements while remote control inputs (arrow keys) are used to change the FOV on TV devices.

The four components described above are the basis for almost all 360° playouts. The difference between existing 360° solutions we discussed in Section 2.4.2 lies in the location where each of these components is running, especially the component for performing the 360° transformation. Therefore, we can identify two classes of 360° streaming solutions: 1) Solutions that rely on Client Side Transformation (CST) and 2) solutions that rely on Server Side Transformation (SST). As we can see in Figure 5.10, CST performs the 360° transformation on the client while SST performs the 360° transformation on the server. In order to guarantee a smooth transition between FOVs, i.e., when the user is equipped with a head-mounted display and moves his head, most solutions rely on CST. In the CST approach, the whole 360° video is streamed to the client, and the 360° transformation is performed locally. Moreover, the CST approach captures and processes user inputs locally, resulting in lower latency compared to the SST approach. Technologies for the CST approach are dependent on target devices and platforms. For cross-device deployments, Web apps are a promising technology to develop code once and reach a variety of devices. Web browsers and HTML5 technology have become a commodity across devices and enable current 360° video solutions with CST. These rely on WebGL (an OpenGL implementation targeted at Web browsers) and the Canvas API. The

W3C WebVR specification uses these APIs to provide support for VR devices such as HMDs. YouTube uses the CST approach and applies ABR (Adaptive Bitrate) on the entire 360° video and uses the same streaming infrastructure as for regular videos. Our measurements have shown that this works well for projected 360° videos at 4K resolution, but for higher resolution 360° videos, CST takes too much time and prevents a smooth transition between FOVs. Besides the processing issue, the bitrate evaluation of YouTube 360° videos in Section 5.3.1 has shown that around 90% of the bandwidth is wasted with streaming unseen content when using the CST approach.

The SST approach addresses the processing and bandwidth issues of the CST approach by performing the 360° transformation on the server instead of the client. As a result, only the FOV video will be streamed to the client and rendered directly to the user similar to regular videos without additional processing on the client. This means that devices with limited capabilities concerning hardware and software resources can be supported as well. The drawbacks of the SST approach are the limited scalability and latency. In SST, the server needs to run an instance of the 360° transformation component for each client which increases the average costs per user. Furthermore, all captured user inputs on the client need to be sent to the 360° transformation component running on the server which increases the latency compared to the CST approach. In Section 5.3.4, we will introduce a novel solution that enables high quality 360° video playback by using pre-processing techniques for preparing the 360° videos in advance and providing a right balance between the CST and SST approaches. Before we introduce the new solution, we will describe the process of generating 16K 360° content which is required to enable 4K FOV.

5.3.3 16K 360° Content Generation

"*360° videos with resolutions higher than 4k are currently rare. However, 16K 360° videos are needed to produce a 4K FOV which can be displayed on 4K screens like UHD TVs. The Blender Foundation and the Google VR team worked together in 2016 to convert the opening sequence of the Llama cartoon "Caminandes" into a 360° VR experience [159]. As a result, they created the 360° equirectangular frames using Blender and generated the 360° video for YouTube in different resolutions up to 8K [160]. Since we need a resolution of at least 16K (4 times 8K) to enable 4K FOV, we generated the 360° equirectangular frames in 16K resolution (16384x8192 pixels) from Blender Caminandes source material*" [21]. Figure 5.5 shows an example of a Caminandes 360° equirectangular frame while Figure 5.6a shows the calculated 4K FOV frame with 90° horizontal FOV angle. It is important to mention that the generation of each equirectangular frame took around 1 hour on a PC with four modern GPUs (NVIDIA's GeForce GTX 1080). In total, we generated 960 equirectangular frames

in PNG format which result in a video duration of 40s with a frame rate of 24fps (960 = 24fps*40s). The generated content will be used to evaluate (in Section 6.3) our pre-rendering based solution by comparing it to existing 360° video streaming solutions. Figure 5.11 shows the difference between a FOV generated from a 4K equirectangular frame and a FOV generated from a 16K equirectangular frame. We can clearly see that the quality of the FOV generated from the 16K equirectangular frame is better than the FOV generated from the 4K frame. This is because a 16K frame has a 16 times higher resolution than a 4K frame as we mentioned earlier.

(a) (b)

Figure 5.11.: (a) FOV created from 4K equirectangular frame vs. (b) FOV created from 16K equirectangular frame

5.3.4 360° Video Pre-rendering Approach

As we discussed in Section 5.3.2, the CST and SST approaches both have advantages and disadvantages. While the CST approach enables low motion-to-photon latency which is a key requirement for 360° playback on HMDs and uses existing streaming infrastructure without the need for computation or graphical processing resources on the server, the SST approach reduces the bandwidth consumption and processing requirements on the client. In this thesis, we will introduce a new 360° streaming and playback solution that provides a good balance between the CST and SST approaches and supports the following requirements:

- reduce bandwidth consumption by streaming only the FOV and not the entire 360° video;
- support constrained devices or any device that can play regular videos without the need for additional processing resources;
- use existing streaming infrastructures and content delivery networks for streaming regular videos;
- increase scalability and reduce operating costs by minimizing additional processing resources required on the server comparing to regular video streaming;
- minimize motion-to-photon latency to a level that enables best user experience depending on the input method and target device;

- support FOV with a native resolution of the target device, i.e., 4K FOV on UHD TVs; and
- support state-of-the-art video codecs especially H.264 which is supported with hardware acceleration on almost any playback device;

The new solution may have some drawbacks such as additional storage compared to the CST and SST approaches which will be evaluated in Section 6.3.
The main idea of this new approach is the pre-rendering of FOV videos in a way that additional processing on the streaming server and the playback device is no longer required. The pre-rendered FOV videos can be stored on streaming servers and delivered to playback devices through existing CDNs without the need to perform the geometrical transformation for calculating the FOV for each connected client. This means that the pre-rendering approach requires more storage resources but less processing resources on the other side. This means that nearly any device that is able to play a video can be supported by this approach. For example, broadcasters can use this solution to offer 360° video streaming on televisions using HbbTV technology at almost the same cost as conventional video streaming. The concept of storing pre-processed content is not new and is already used in the media streaming domain, especially for adaptive bitrate streaming. In this case, the source video will be pre-processed, and multiple versions of it will be generated and stored for different combinations of bitrate, resolution, and codec. This allows the player on the client to select the best suitable version of the video depending on available bandwidth, display resolution and supported video codecs.
In our 360° streaming approach, we will pre-render and store multiple FOV videos with certain overlap by varying the view angle along the horizontal and vertical axis in the spherical space. The overlap factor has an impact on the number of FOV videos and the navigation granularity which will be discussed in more detail in this section. The motion-to-photon latency is one of the critical factors that has a direct impact on usability. In the case of head-mounted displays, the maximum allowed delay is 20ms to avoid motion sickness. Our solution cannot reduce the latency to 20ms thus is not suitable for HMDs. But for flat screen devices like TVs, it offers a solution with a unique user experience that allows viewer to display 360° videos and use the TV remote control for navigation without the need for additional hardware. Bringing the 360° video experience to the TV is what many content providers, especially broadcasters, are currently looking for. HbbTV is the enabler technology that makes our solution attractive to broadcasters. As already mentioned, most German and many European broadcasters already offer HbbTV services such as electronic programme guide (EPG) and video-on-demand (VOD) services. With our solution, broadcasters can expand this offering with a 360° video playback service that can be easily integrated into existing HbbTV applications. The architecture of our approach is shown in Figure 5.12 and comprises four steps: *Pre-processing*, *Storage*, *Streaming*, and *Playback*. The pre-processing step includes the pre-rendering

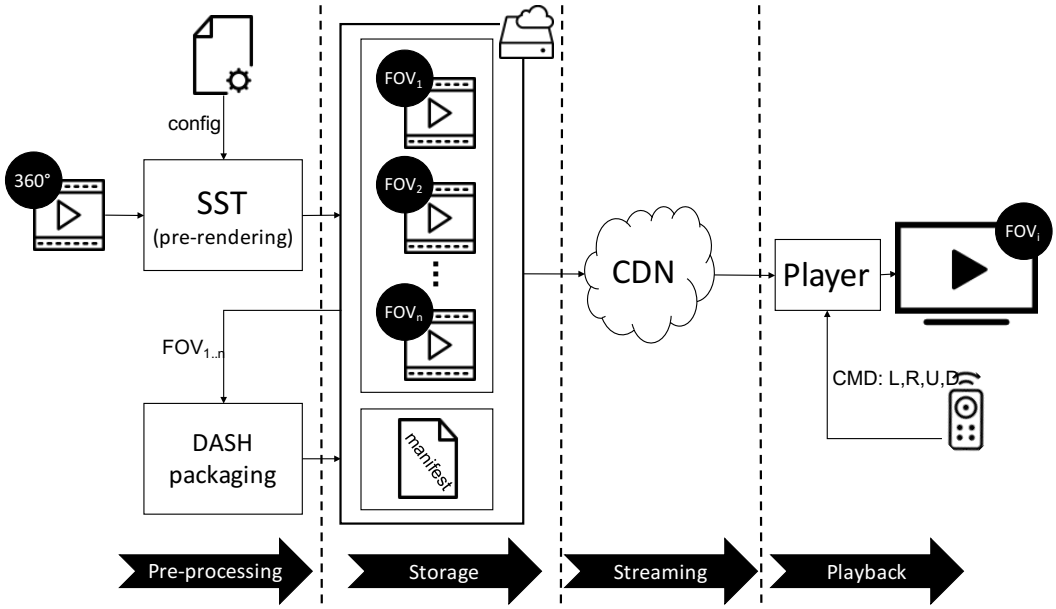


Figure 5.12.: 360° Video Pre-rendering Approach

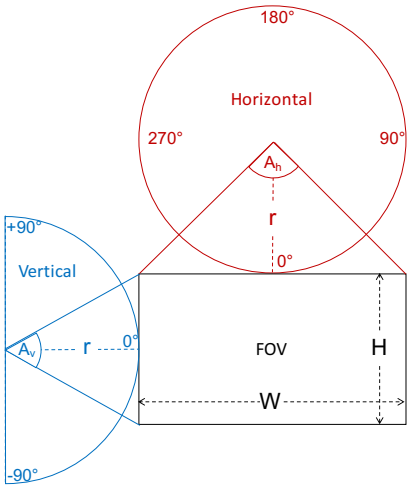
and packaging of all FOV videos which will be stored on dedicated streaming servers in the next step. Afterwards, the created FOV videos and manifest files will be made available for clients through existing CDNs. The manifest file provides all the configurations and locations of the FOV video segments together with other relevant information for the player. Section 5.3.6 provides more information about the manifest file and its structure. In the last step "*Playback*", the client requests the manifest file of a video which includes information about pre-rendered FOVs, starts playback with the default FOV and reacts to user inputs to change the FOV. The four steps are described in detail in the following subsections.

Pre-processing

The pre-processing step includes the two components "*Pre-renderer*" and "*Packager*" that will be described in this subsection. The pre-renderer operates on the source 360° video and calculates the requested FOVs defined in the configuration file provided as input. A FOV is defined using the four parameters (ϕ, θ, A_h, A_v) where:

- A_h is the horizontal opening angle of the FOV in degree
- A_v is the vertical opening angle of the FOV in degree
- ϕ is the horizontal angle in degree measured from the origin to the center of the FOV. $0° \leq \phi < 360°$
- θ is the vertical angle in degree measured from the origin to the center of the FOV. $(-90° + \frac{A_v}{2}) \leq \theta \leq (90° - \frac{A_v}{2})$

The opening angle (A_h, A_v) defines the zoom level of the FOV and remains constant during pre-rendering if only a single zoom level is requested. Most 360° video players like YouTube provide a default FOV with A_h between 90° and 100° and $16:9$ aspect ratio. In our case, we will use a default vertical opening angle of $A_v = 60°$ and a $16:9$ aspect ratio which results in a horizontal opening angle of $A_h = 91.5°$. Figure 5.13 and the corresponding equations 5.1-5.6 explain the relationship between A_h and A_v and how $A_h = 91.5°$ is calculated. The FOV is the projection of a part



$$\frac{W}{H} = \frac{16}{9} \tag{5.1}$$

$$\tan(\frac{A_h}{2}) = \frac{W}{2} * \frac{1}{r} \tag{5.2}$$

$$\tan(\frac{A_v}{2}) = \frac{H}{2} * \frac{1}{r} \tag{5.3}$$

$$\frac{\tan(\frac{A_h}{2})}{\tan(\frac{A_v}{2})} = \frac{16}{9} \tag{5.4}$$

$$A_h = 2 * tan^{-1}(\frac{16}{9} * \tan(\frac{A_v}{2})) \tag{5.5}$$

$$A_v = 60° \Rightarrow A_h = 91.5° \tag{5.6}$$

Figure 5.13.: FOV with a WxH resolution and aspect ratio 16:9

of the sphere onto the tangent plane at the point ϕ and θ. The dimension of the FOV depends on the opening angles A_h (Equation 5.2) and A_v (Equation 5.3). A_h determines the height H of the FOV while A_v determines the width W of the FOV. Since we need a specific aspect ratio, e.g., 16:9 (Equation 5.1), we need to pass only the value of A_v or A_h and the value of the second parameter can be calculated using Equation 5.5.

We will consider in the remainder of this section a constant zoom level (constant FOV opening angle (A_h, A_v)) and use (ϕ, θ) as a shortcut to describe a FOV instead of (ϕ, θ, A_h, A_v). In general, a single zoom level is sufficient for most use cases, especially for TV. If multiple zoom levels are required, the FOV pre-rendering algorithm must be applied to each zoom level (A_h, A_v).

Since the idea of our approach is to pre-render FOVs in advance, it is important to know the angle of each of these FOVs. One way to do this is to specify a constant horizontal and vertical angle distance between two adjacent FOVs. The horizontal angle distance between two adjacent FOVs (ϕ_1, θ) and (ϕ_2, θ) is defined as $\Delta\phi$, and the vertical angle distance between two adjacent FOVs (ϕ, θ_1) and (ϕ, θ_2) is defined as $\Delta\theta$. In other words, all adjacent FOVs of (ϕ, θ) along the x-axis and y-axis are: $(\phi - \Delta\phi, \theta)$, $(\phi + \Delta\phi, \theta)$, $(\phi, \theta - \Delta\theta)$ and $(\phi, \theta + \Delta\theta)$. $\Delta\phi \leq A_h$ or $\Delta\theta \leq A_v$ means that there is an overlap between the adjacent FOVs. Figure 5.14 shows an example for the FOVs by varying (ϕ, θ) with $\Delta\phi = 30°$ and $\Delta\theta = 30°$. The number of

FOVs for each horizontal level (keeping vertical FOV angle θ constant and changing horizontal FOV angle ϕ stepwise by $\Delta\phi$) is $N_h = \frac{360°}{\Delta\phi}$ and the number of FOVs for each vertical level (keeping horizontal FOV angle ϕ constant and changing vertical FOV angle θ stepwise by $\Delta\theta$) is $N_v = \frac{180°}{\Delta\theta} - 1$. The total number of FOVs is then $N = N_h * N_v$. Figure 5.14 shows all combinations of FOV angles ϕ and θ with the following settings:

- $A_h = 91.5°$ and $A_v = 60°$ (A_h is calculated from A_v as shown in Figure 5.13)
- $\Delta\phi = 30°$ and $\Delta\theta = 30°$
- $0° \leq \phi < 360°$ and $-60° \leq \theta \leq 60°$ ($-60° = -90° + \frac{A_v}{2}$ and $60° = 90° - \frac{A_v}{2}$)
- $N_h = \frac{360°}{\Delta\phi} = 12, N_v = \frac{180°}{\Delta\theta} - 1 = 5, N = N_h * N_v = 60$

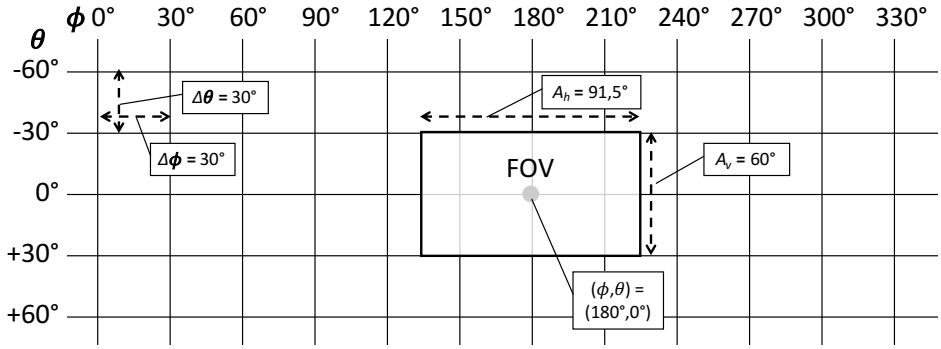


Figure 5.14.: FOVs by varying ϕ and θ stepwise with $\Delta\phi = 30°$ and $\Delta\theta = 30°$

The main steps for pre-rendering all FOVs that come into account are shown in the main function of Algorithm 3. In the first step, the function SETUPFOVANGLES( ) calculates and adds all combinations of FOV angles to the list $FOVAngles$ based on the parameters $\Delta\phi$ and $\Delta\theta$. In the next step, RENDERFOVFRAMES( ) renders the FOV frames for each angle in $FOVAngles$ and each equirectangular frame $EQRFrame_i$ decoded from the source 360° video. The configuration of the example in Figure 5.14 was used in most of our pre-rendered videos. Furthermore, the process can be optimized if less or non-relevant FOVs are skipped and not rendered. This is the case, for example, when the relevant regions of interest are located around the equator of the equirectangular video (vertical FOV angle $\theta = 0°$). As a result, $\Delta\theta = 60°$ (no vertical overlap) can be used instead of $\Delta\theta = 30°$ (vertical overlap factor = 50%) with no major impact on the user experience. In this case, the total number of FOVs $N = N_h * N_v$ can be reduced from 60 ($N_h = 12$ and $N_v = 5$) to 36 ($N_h = 12$ and $N_v = 3$). It is also possible to skip non-relevant FOVs, i.e., for $\theta \neq 0°$ and render only relevant FOVs, e.g., for $\theta = 0°$. In this case, the total number of FOVs N can be reduced to 12 ($N_h = 12$ and $N_v = 1$). This setting was applied in many pre-rendered 360° videos especially those recorded using a static 360° camera (position of the camera does not change during the recording). Figure

5.15 shows a snapshot of a 360° video provided by the German public broadcaster ZDF during the Biathlon World Cup in Oberhof/Germany from January 10 to 13, 2019[161]. As we can see, the upper and lower parts of the image can be skipped since the relevant regions of interest are located in the middle part of the image. ZDF used our solution with this configuration and offered a 360° live streaming of the Biathlon World Cup via HbbTV [162]. It is important to mention that projections

Figure 5.15.: Snapshot of a 360° video frame during the Biathlon World Cup 2019 in Oberhof/Germany

other than equirectangular can be applied without changing the main algorithm. Only the function RENDERFOVFRAME( ) needs to be updated to create a FOV frame using the new projection. In the last step, the function CREATEFOVSEGMENTS( ) creates the FOV video segments for each combination of angle (ϕ, θ) from the FOV frames created in the previous step. The parameter GOP specifies the number of frames in a video segment. The acronym GOP stands for Group Of Pictures which is an important parameter for encoding a video and has an impact on the compression ratio which may vary depending on the used video codec. GOP has also an impact on the latency when switching between different FOVs. This happens because the first frame of a segment is a self-contained picture (Intra-coded picture or I-frame) that can be independently decoded and displayed, and all other frames in the same segment can be predicted only from the previous frame (Predicted Picture or P-frame) or from the previous and the following frames (Bidirectional predicted picture or B-frame) [163]. Since B-frames are predicted from past and future frames (I/P-frames) in the same GOP, the video decoder needs to load future I/P-frames in order to decode the B-frames. In other words, the video player can only start the playback from the first frame (I-Frame) of the new segment and therefore, a switch between FOVs without skipping frames is only possible after the playback of the current segment is completed. The duration of a segment can be calculated as $\frac{GOP}{FPS}$ seconds, for example, a segment with GOP size of 10 frames and frame rate

of 50 frames per second results in a segment duration of 0.25 seconds or 250ms. In this case, the average latency caused by the segment duration is 125ms. In Section 6.3, we will compare our solution to other existing solutions by evaluating them according different metrics including the latency.

Algorithm 3 Pre-rendering of all FOVs

Input: A_h, A_v, W, H ▷ FOV dimension
Input: $EQRVideo$ ▷ Input Equirectangular 360° video
Input: FPS ▷ Video frame rate
Input: N ▷ Total number of video frames
Input: GOP ▷ Number of frames in a FOV video segment
Define: $EQRFrame_i$ ▷ Equirectangular 360° frame i
Define: $FOVFrame_{i,\phi,\theta}$ ▷ FOV frame i for angle (ϕ, θ)
Define: $FOVSegment_{j,\phi,\theta}$ ▷ FOV video segment j for angle (ϕ, θ)
Define: $FOVAngles \leftarrow \{\}$ ▷ All combinations of FOV angles (ϕ, θ)

function MAIN() ▷ The start function of the algorithm
  SETUPFOVANGLES( )
  RENDERFOVFRAMES( )
  CREATEFOVSEGMENTS( )
end function

function SETUPFOVANGLES() ▷ Calculates all combinations of FOV angles (ϕ, θ)
  $\phi \leftarrow 0°$
  while $\phi < 360°$ **do**
    $\theta \leftarrow 0°$
    $FOVAngles \leftarrow FOVAngles \cup (d, \phi, \theta)$
    while $\theta \leq 90° - \frac{A_v}{2}$ **do**
      $\theta \leftarrow \theta + \Delta\theta$
      $FOVAngles \leftarrow FOVs \cup (\phi, +\theta)$
      $FOVAngles \leftarrow FOVs \cup (\phi, -\theta)$
    end while
    $\phi \leftarrow \phi + \Delta\phi$
  end while
end function

```
19: function RenderFOVFrames()        ▷ Generates FOV frames from EQR frames
20:     i ← 0                                          ▷ Index of current frame
21:     while i < N do
22:         EQRFrame_i ← DecodeNextFrame(EQRVideo)
23:         for all (φ, θ) in FOVAngles do
24:             FOVFrame_{i,φ,θ} ← RenderFOVFrame(EQRFrame_i, φ, θ, A_h, A_v, W, H)
25:         end for
26:         i ← i + 1
27:     end while
28: end function

29: function CreateFOVSegments() ▷ Generates FOV Segments from FOV frames
30:     j ← 0                                          ▷ Index of current segment
31:     while j * GOP < N do
32:         EQRFrame_i ← DecodeNextFrame(EQRVideo)
33:         for all (φ, θ) in FOVAngles do
34:             i1 ← j * GOP
35:             i2 ← (j + 1) * GOP
36:             FOVSegment_{j,φ,θ} ← CreateFOVSegment(FOVFrame_{i1..i2,φ,θ}, GOP, FPS)
37:         end for
38:         j ← j + 1
39:     end while
40: end function
```

Storage

After all FOV video segments have been rendered in the previous step, they are made available on a simple file storage server following a specific file and folder structure. There are several methods for storing the FOV video segments. The two most important are:

Method 1: Each FOV video segment is saved in a separate video file. All video files related to segments of the same FOV are grouped in a folder with an appropriate name. Furthermore, a manifest file which holds information about existing video segments like file path of each FOV segment will be created and stored in the root folder of the video. The example below shows the file and folder structure of this method.

```
root
├── fov-0-0
│   ├── seg-0.mp4
│   ├── seg-1.mp4
│   └── ...
├── fov-30-0
│   ├── seg-0
│   ├── seg-1
│   └── ...
├── ...
└── manifest.json
```

Method 2: All video segments related to a FOV angle (ϕ, θ) are stored in the same order in a single file. In order to locate a segment in the corresponding FOV video, the byte offset of the segment in the file must be known. Therefore, the manifest file which is also available in the root folder as in the first method needs to hold the byte offset of each FOV segment. The example below shows the file structure of this method.

```
root
├── fov-0-0.mp4
├── fov-30-0.mp4
├── ...
└── manifest.json
```

Each of the storage methods described above has its advantages and disadvantages. In the first method, the manifest file is very compact since there is no need to store the byte offsets for each segment as in the second method. On the other hand, the second option allows the client to request multiple segments in a single HTTP request by using the *HTTP Range* header and reduce the overhead to establish an HTTP connection for every single segment. The second method performs better if the player runs in a browser that supports the W3C Fetch API [52]. This API allows the application to access chunks of the binary data sent in the HTTP response while the content is still downloading. In this case, the player can request multiple segments from the CDN in a single HTTP request and append each segment to the video buffer without the need to wait for all requested segments to be downloaded. If the Fetch API is not supported, each segment needs to be downloaded in a separate HTTP request using the old XHR API [51] which allows to access the content after all data is received and the connection is closed. In this case, the first method performs better since the manifest file is much smaller and simpler to parse than in the second method.

Streaming

After the video segments and manifest files are available on the storage server, they can be streamed to the client. The client decides based on user inputs which segment to request at which time. There are two streaming approaches that can be applied (Figure 5.16): In the first approach, a streaming server which acts as a proxy

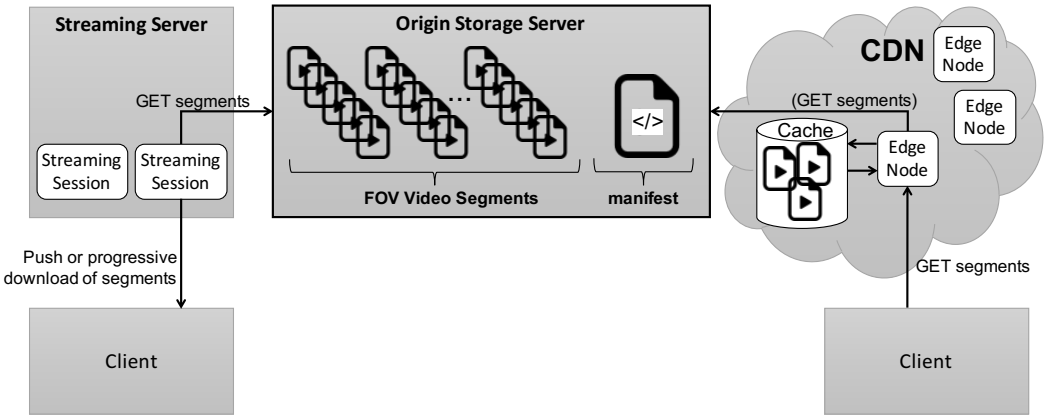


Figure 5.16.: 360° Streaming Approaches

between the storage server and the client is required. For each connected client, the streaming server creates a session which holds necessary information like current FOV angle, segment index, and other relevant information. The streaming server pushes new segments to the corresponding connection. The second approach uses Content Delivery Networks (CDNs) which enable stateless connections to one or multiple segment files and allow to run the entire streaming logic on the client. The segments will be cached on dedicated edge nodes in the CDN. The CDN approach has proven to be the most effective method and is the de-facto standard for media streaming over the Internet. The first approach can be applied to legacy devices like old HbbTV terminals that are not able to construct the final video stream from the individual FOV video segments due to the missing APIs to control and manage the video buffer. The next section will describe the player components for the second approach.

Playback

The player constructs the final video stream from the individual FOV video segments. There is no need to process the received video data before playback. The client platform only needs to provide an API that allows the application to control the video buffer by adding, removing or replacing video segments to or from it. In our implementation, we focused on Web technologies and used the W3C Media Source

Extension API (MSE) for this purpose. The player consists of the following three components:

- **Manifest parser**: The URL of the manifest file is the only input required in the player. As described above, the manifest contains all metadata of the video as well as all information of the available FOVs and the bytes offsets for each FOV video depending on which storage method is used. The implementation section shows how to use the DASH Media Presentation Description (MPD) as a manifest format for this solution. A Web client can request the manifest file using a simple HTTP GET request.
- **Player and Buffer Control**: After the manifest is parsed, the player will be initialized with the default FOV angle (0, 0). By default, the player starts the playback from segment index $j = 0$. In each step, the player requests a segment from the server using an HTTP GET request. If the segments corresponding to a FOV angle are stored in a single file (as described in storage method 2 above), then the client needs to determine the byte offset of the first and last segment from the manifest file in order to calculate the value of the HTTP Range header.
- **User Input Control**: This module is responsible for navigating in the video. It receives a request from the input device, for example, a TV remote control or keyboard and changes the FOV. The player updates its internal state with the coordinates of the new FOV and sends a new HTTP GET request to retrieve the next segments of the new FOV. Once the new segments are received, the segments of the old FOV will be automatically replaced.

5.3.5 Improvement

The pre-rendering-based approach we introduced in this thesis has a drawback concerning the transition between FOVs: If the viewer changes the FOV, i.e., using the arrow keys of the TV remote control, then the video segments of the target FOV will be requested and appended to the video buffer. This leads to an abrupt transition between the two perspectives, which has a negative impact on usability and does not give the viewer the feeling of navigating in a 360° video. This happens because only static FOVs for selected perspectives are rendered. Figure 5.17 illustrates this problem. As we can see, the FOV $\phi = 15°$ is replaced by the adjacent FOV $\phi = 30°$ at time $t = 1s$. The viewer will not see the FOVs between $\phi = 15°$ and $\phi = 30°$. To solve this issue, we improved the solution by rendering transition videos (also called motion videos) in addition to the static FOV videos. The number of the transition videos depends on the directions that must be supported. When the four arrow keys of TV remote control are considered, then four transition videos (left, right, up and down) are needed for each static FOV video. This increases the total number of

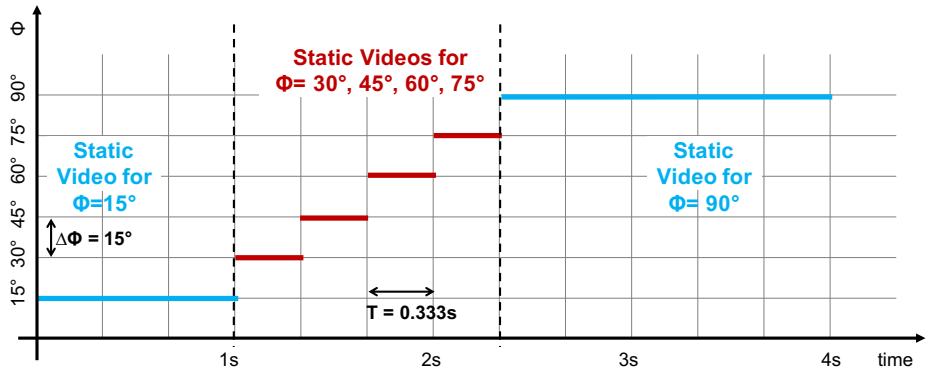


Figure 5.17.: Abrupt transition between FOVs

videos that must be rendered to $N = 5 * N_h * N_v$ (static, left, right, up and down). For example, the transition video $FOV_L(\phi, \theta)$ with left motion starts at video time

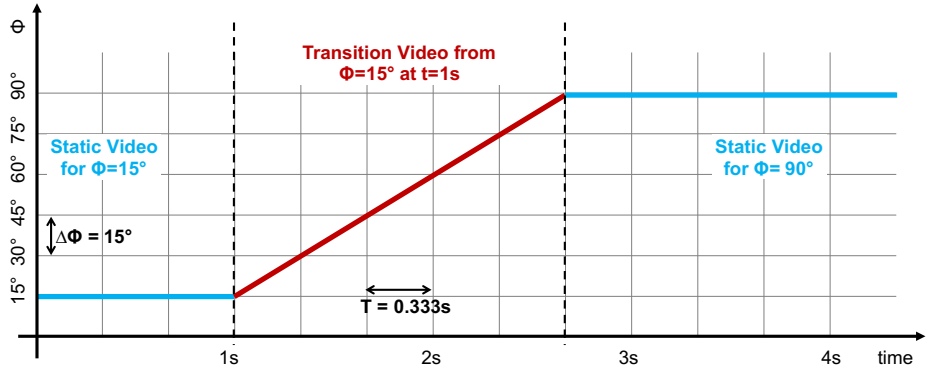


Figure 5.18.: Dynamic transition between FOVs

$t = 0$ by FOV (ϕ, θ) while the horizontal angle ϕ increases by $\Delta\phi$ within the period T ($T = \frac{GOP}{FPS}$ is the duration of a FOV segment). This means that at time $t = T$, the transition video reaches the FOV angle $(\phi + \Delta\phi, \theta)$ and at time $t = 2 * T$ it reaches the FOV angle $(\phi + 2 * \Delta\phi, \theta)$ and so forth. The example in Figure 5.18 shows the transition video with left motion from angle $\phi = 0°$ at time $t = 1s$ to angle $\phi = 90°$ at time $t = 2.666s$ for $\Delta\phi = 15°$, $GOP = 10$ and $FPS = 30$ which results in segment duration of $T = 0.333s$.

The idea of the improvement by rendering transition videos is submitted in November 2017 to the *German Patent And Trade Mark Office*[1], and the patent "[DE] Verarbeitungsverfahren und Verarbeitungssystem für Videodaten" (*DE102017125544B3*) [164] has been granted and published in June 2018. An international application for the invention is also submitted in April 2018 to the *World Intellectual Property Organization - WIPO*[2] and the international patent "Processing method and processing system for video data" (*WO2018210485A1*) [165] has been published in November 2018.

[1] https://www.dpma.de
[2] https://www.wipo.int

5.3.6 Implementation

In order to evaluate the pre-rendering approach and compare it to other existing solutions (see evaluation section), we implemented it including the components described in the previous section as a proof-of-concept prototype that runs on Amazon AWS. An overview of the technologies used in the implementation is provided in Figure 5.19. For FOV rendering, we selected Amazon EC2 G3 instances that provide

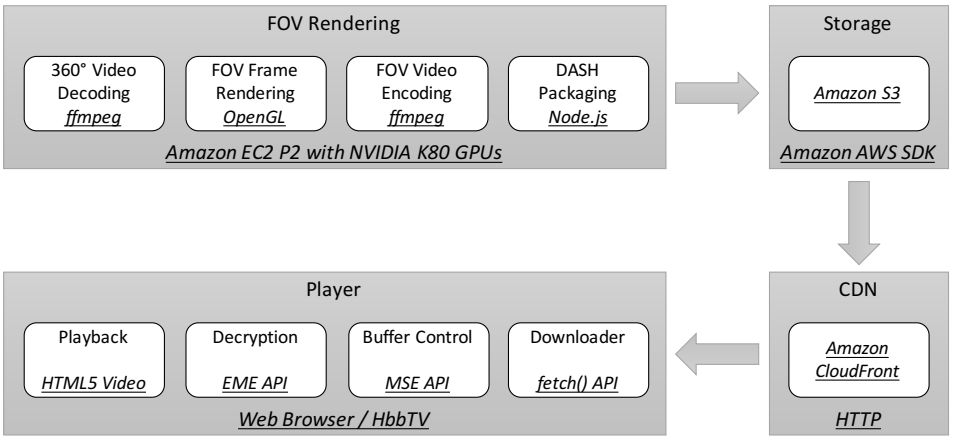


Figure 5.19.: Implementation Technology Stack

GPU-based (NVIDIA K80 GPUs) parallel compute capabilities[3] which are required in our case to perform the 360° transformation. The prototype supports as input equirectangular videos in any format that can be decoded by *ffmpeg*. Afterwards, the FOV frames for all combinations of angles (ϕ, θ) are calculated from each equirectangular frame using OpenGL, a cross-platform library for 2D and 3D graphics. The FOV frames will be then encoded into FOV video segments also using *ffmpeg*. In order to guarantee playback interoperability across devices, we choose MPEG DASH [67] as the streaming format and ISOBMFF as the file format. To enable quick switching between FOVs on the client side, low latency streaming mechanisms are utilized. After all FOV ISOBMFF video segments and DASH manifest are generated, they will be uploaded to Amazon Simple Storage Service S3[4] without changing the file and folder structure. For delivery, we used Amazon's CDN CloudFront[5] that can be configured easily to use AWS S3 as an origin for media files. Other CDNs like Akamai can be used instead of CloudFront. Listing 5.4 shows an example of the DASH manifest where each FOV is described as a separate *AdaptationSet*.

```
<?xml version='1.0' encoding='utf-8'?>
<MPD availabilityStartTime="2017-05-11T14:29:14.667Z"
     publishTime="2017-05-11T14:29:14.667Z"
     maxSegmentDuration="PT0.5S"
     mediaPresentationDuration="PT1M14S"
```

[3] https://aws.amazon.com/ec2/instance-types/g3/
[4] https://aws.amazon.com/s3/
[5] https://aws.amazon.com/cloudfront/

```
     minBufferTime="PT4S"
     profiles="urn:mpeg:dash:profile:isoff-live:2011">

  <Period id="0" start="PT0S">
    <!-- AdaptationSet for FOV (static,0,0) -->
    <AdaptationSet codecs="avc1.64001F" contentType="video" mimeType="video/mp4"
     id="static-0-0">
      <Role schemeIdUri="urn:mpeg:dash:role:2011" value="main"/>
      <SupplementalProperty schemeIdUri="urn:fhg:fokus:fov:2017" value="(static
     ,0,0)"/>
      <SegmentTemplate duration="0.5" initialization="$RepresentationID$/init.mp4"
                       media="$RepresentationID$/seg-$Number$.m4s" startNumber="1"
      />
      <Representation bandwidth="5000000" id="video-5000000-static-0-0" />
      <Representation bandwidth="2000000" id="video-2000000-static-0-0" />
      <Representation bandwidth="1000000" id="video-1000000-static-0-0" />
    </AdaptationSet>

    <!-- AdaptationSet for FOV (static,30,0)-->
    <AdaptationSet codecs="avc1.64001F" contentType="video" mimeType="video/mp4"
     id="static-30-0">
      <Role schemeIdUri="urn:mpeg:dash:role:2011" value="main"/>
      <SupplementalProperty schemeIdUri="urn:fhg:fokus:fov:2017" value="(static
     ,30,0)"/>
      <SegmentTemplate duration="0.5" initialization="$RepresentationID$/init.mp4"
                       media="$RepresentationID$/seg-$Number$.m4s" startNumber="1"
      />
      <Representation bandwidth="5000000" id="video-5000000-static-30-0" />
      <Representation bandwidth="2000000" id="video-2000000-static-30-0" />
      <Representation bandwidth="1000000" id="video-1000000-static-30-0" />
    </AdaptationSet>

    <!-- other AdaptationSets for remaining FOV videos -->
  </Period>
</MPD>
```

Listing 5.4: Example DASH Manifest with FOV AdaptationSets

"Using *SupplementalProperty*, the FOV type (static or motion) and FOV angle are described. Within each *AdaptationSet*, multiple *Representations* with varying bitrates of the FOV video are made available. An *AdaptationSet* includes a role element with value *"main"* and all other *AdaptationSets* include a role element with value *"alternate"*. The value of the role element is used in the DASH player to select the default FOV. Since the transition between FOVs is triggered by the user, i.e., using remote control inputs, existing DASH players need to be extended to implement the transition logic by selecting the appropriate *AdaptationSet*. An *AdaptationSet* may also contain a *"low-latency"* representation, which has higher bitrate due to short segment length and is used when the player switches between FOVs. Also, it can contain a *"regular"* representation with longer segment lengths, e.g., 2s which is used by the player when the FOV remains unchanged. This representation saves bandwidth, because of lower bitrates due to longer segment lengths" [21]. After the packaging is completed, all FOVs and the manifest file are published to Amazon's

CDN CloudFront which can be easily configured to use AWS S3 as CDN origin. Other CDNs like Akamai can be used instead of CloudFront as well.

"On the client side, we leverage Web technologies such as W3C Media Source Extension API (MSE). MSE API allows Web applications to control the source buffer of an HTML5 video object by appending, removing or replacing segments. No Canvas API is needed since pre-rendering was used in the previous step. Therefore, the content can also be DRM-protected and played with the help of the W3C Encrypted Media Extensions API (EME). We use a single MSE *SourceBuffer* for seamless transitions between FOVs. Multiple *SourceBuffers* could cause video decoding interrupts. Using MSE's *appendBuffer()* ISOBMFF segments of a FOV are fed into the SourceBuffer for playback. When the FOV changes, existing segments are replaced by segments of the new FOV. Moreover, the adaptation logic in a DASH player needs to be modified for this type of playback. Besides pre-buffering of adjacent FOVs, the different bitrate representations can be used to optimize FOV switching latency further. For example, when the user is switching between FOVs, only the lowest bitrate of the low-latency Representations is requested from the CDN. Once the FOV remains unchanged, and the playback stabilizes, the adaptation logic can decide to switch to higher bitrates. Furthermore, for requesting FOV video segments we use the new W3C fetch API instead of XHR API. The fetch API allows the client to access downloaded chunks before the whole content is fully loaded. In this case, the player can request multiple segments in a single request (using the HTTP Range header) and still be able to access each segment as soon as all its chunks have been downloaded." citeBass1804:Streaming

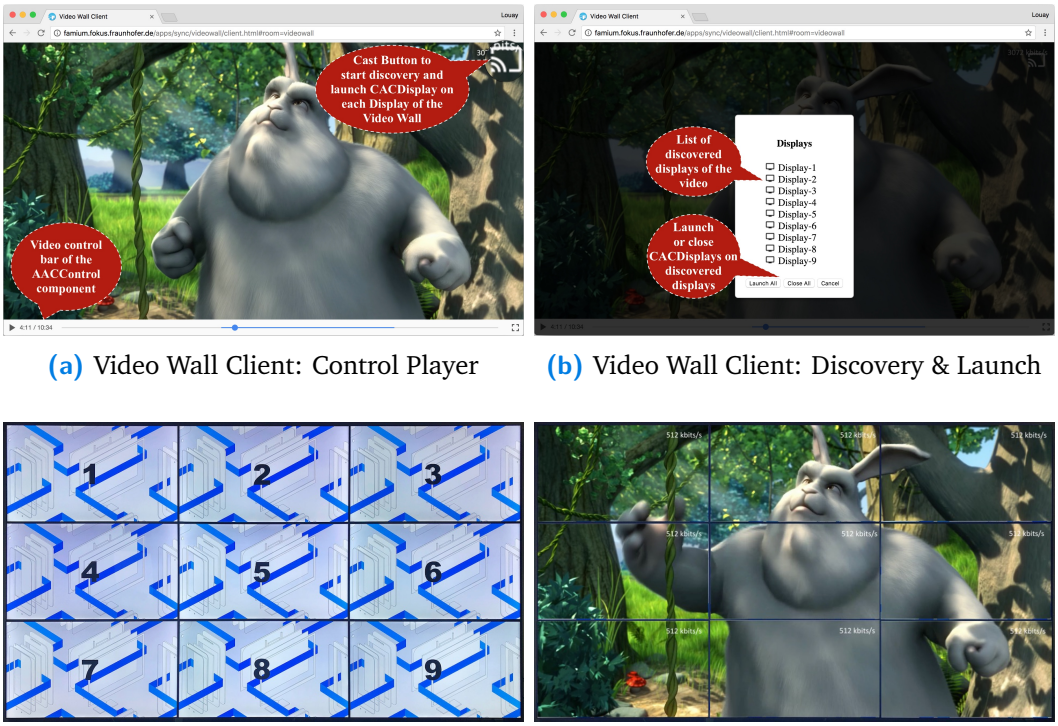
6 Evaluation

In this chapter, we will evaluate the approaches and solutions presented in this thesis and compare them with existing state-of-the-art solutions. Section 6.1 evaluates the Multiscreen Application Model and the Media Synchronization algorithm while Section 6.2 evaluates the three Application Runtime approaches introduced in Section 4.4.1 according to different metrics like bandwidth, latency, and battery life. Finally, Section 6.2.4 provides an evaluation of the 360° pre-rendering approach we introduced in Section 5.3 and compares it to existing state-of-the-art rendering approaches.

6.1 Multiscreen Application Model and Media Synchronization

In order to evaluate the accuracy of the Multistream synchronization algorithm we introduced in Section 5.2.2, we developed a prototype based on the Multiscreen Application Model that implements the video wall use case described in Section 3.1.5. Besides the evaluation of the synchronization accuracy, this use case addresses also most of the identified multiscreen requirements listed in Section 3.2.1 such as discovery, launch, instantiation, communication, terminating and joining. From the use case defined in Section 3.1.5, which describes the functionality of the video wall, we can identify the two composite application components *CACClient* and *CACDisplay*, which in turn include the atomic components *AACControl* and *AACPlayer*, as described below:

- **Atomic Application Component *AACControl*:**
  - As depicted in Figure 6.1a, this component provides a cast button that allows the user to discover displays of the video wall;
  - launches a *CACDisplay* instance on each discovered display. For the sake of simplification, we assume that the names of the displays are used to determine the position of the corresponding display in the video wall as depicted in Figures 6.1b and 6.1c;
  - assigns a video URL to each *AACPlayer* instance and uses the display names to determine the video URL of the corresponding tile;



(a) Video Wall Client: Control Player (b) Video Wall Client: Discovery & Launch

(c) Video Wall Display: Before Launch (d) Video Wall Display: After Launch

Figure 6.1.: Video Wall Application Components

  - provides player controls like "play", "pause", and "seek" that allow the user to control the playback of the video wall.

- **Atomic Application Component *AACPlayer*:**

  - runs inside *CACClient* or *CACDisplay* components;
  - receives target video URL from the *AACControl* instance;
  - all player instances are kept in sync by assigning each of them to the same sync group.

The implementation of the Video Wall Multiscreen application is described in Section B.2.2 of the Appendix, and the corresponding Multiscreen Model Tree is depicted in Section B.2.1. The Multiscreen Model Tree captures the status of the video wall application during all relevant phases at runtime. Visualizing the whole application state in a single model makes the development of the application much easier since the application components are derived directly from it. Each of the identified atomic and composite components is implemented as a Web Component following the approach we introduced in Section 4.5. The advantages of this approach are summarized below:

- the application is built using modular and reusable components (atomic or composite).

- the introduced approaches and concepts hide the complexity of integrating individual multiscreen features by using simple and powerful APIs.
- it is built on top of standardized web technologies which are essential for developing interoperable multiscreen applications since the involved devices may run different platforms and operating systems while most of them provide a web browser or embedded web runtime.
- enables migration of application components between devices without noticeable effort;
- enables synchronization of application content and media streams across different devices through a simple API that implements the synchronization algorithm introduced in this thesis;
- allows using different application concepts and approaches introduced in Section 4.3 in the same application without changing the code. This enables a high degree of flexibility by selecting the best suitable approach for a concrete application;
- supports multiple application distribution methods that can be applied based on available computing resources and media rendering capabilities on each connected device; If target devices are unable to process or render the application, the processing-intensive components or the entire application can be migrated to dedicated servers in the cloud without modifying or updating the application.

Content Generation for the Video Wall Application: For evaluation purposes, we set up a video wall with nine displays in a 3x3 matrix. Each of the displays has a resolution of 1920x1080 pixels which results in a total resolution of 5760x3240 pixels. The computer-animated film *Big Buck Bunny* by the Blender foundation [166] is used as input content for the Video Wall. The source video was made using the Blender software and is available under the Creative Commons License Attribution 3.0. For the video wall application, we need to split the content into nine tiles (3x3 matrix) that can be mapped to the displays of the video wall. We used the open source software *ffmpeg* for this purpose. The following *ffmpeg* commands are used to generate the 9 video tiles from the source video *bbb.mp4* (video/audio codec is H.264/AAC):

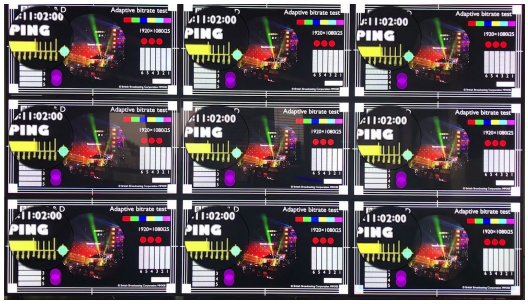
```
$ ffmpeg -i bbb.mp4 -filter:v "crop=in_w/3:in_h/3:0:0" bbb-1.mp4
$ ffmpeg -i bbb.mp4 -filter:v "crop=in_w/3:in_h/3:in_w/3:0" bbb-2.mp4
  ...
$ ffmpeg -i bbb.mp4 -filter:v "crop=in_w/3:in_h/3:in_w*2/3:in_h*2/3" bbb-9.mp4
```

In next step, the DASH content for each video tile (*bbb-1.mp4* ... *bbb-9.mp4*) will be generated and made available on a static HTTP server or CDN. The same configuration which includes five bitrate levels (0.5 Mbps, 1Mbps, 1.5Mbps, 2Mbps, and 3Mbps) and H.264/AAC as video/audio codec is used for the source video

and all tiles. The DASH content is generated using *node-segmenter* developed at Fraunhofer FOKUS. *node-segmenter* is a command line tool written in *Node.js* which generates DASH compliant content from various input sources (Meanwhile *ffmpeg* supports also DASH as output format which can be used to create the DASH content in a single command):

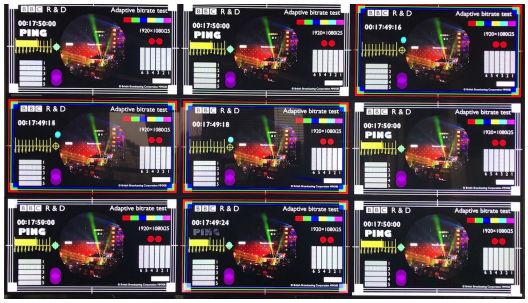
```
$ node-segmenter -i bbb.mp4 -c config.json bbb/manifest.mpd
$ node-segmenter -i bbb-1.mp4 -c config.json bbb-1/manifest.mpd
  ...
$ node-segmenter -i bbb-9.mp4 -c config.json bbb-9/manifest.mpd
```

The most relevant part of the distribution logic of the video wall application is provided in the *AACControl* component shown in Listing B.6 of Appendix B.2.2. It uses the APIs for discovery, connecting, disconnecting, launch and communication in one place without increasing the complexity of the application. This allows the developer to focus on the essentials for implementing the application itself and frees him/her from common implementation details that occur in nearly every multiscreen application and can be provided by the underlying platform. For example, in the video wall application the synchronization of all video tiles is implemented in the *AACPlayer* component in Listing B.7 in just two lines of code (*lines 14-15*) by assigning the video element on each device to the *SyncGroup* with the same name *VideoWall*. The screenshots of the video wall application components depicted in Figure 6.1 show the video wall using the *Big Buck Bunny* content created as described above. However, for evaluation purposes, it is difficult to measure the synchronization accuracy of the playback on the displays using this content. Instead, we will use test streams provided by the BBC Research and Development group [167] which are created as MPEG DASH test streams and to measure the synchronization accuracy between multiple players since each video frame contains indicators like time and color codes that can be used to uniquely identify the current frame and playback time. In order to capture the playback on all displays of the video wall, the recording should be made using a camera with high frame rate to achieve a better precision. For example, if the camera used for the recording has a frame rate of 60 FPS (frames per second), then we can achieve a precision of $16.67ms$ (time between two adjacent frames). The snapshots of the recording in Figure 6.2 show the video wall at four different playback times. For example, Figure 6.2a shows the video wall at video time *00:11:02:00* where we can see that all displays are presenting the same frame if we compare the time codes. It is important to mention that the time code has the format *hh:mm:ss:ff* where *ff* is the index of the video frame in the current second *ss*. The video on each display has a resolution of *1920x1080* pixels and a frame rate of 25 frames per second which results in frame indexes *ff* ranging from *00* to *24* (first and last frames in a second).



(a) Video Wall Snapshot at 00:11:02:00

(b) Video Wall Snapshot at 00:17:48:17



(c) Video Wall Snapshot at 00:17:50:00

(d) Video Wall Snapshot at 00:17:58:00

Figure 6.2.: Video Wall Components

In order to measure the synchronization accuracy very precisely at any time during playback, we started the video wall application using the BBC test video as input and recorded all displays using a camera with higher frame rate than the test video itself. In our evaluation, we used the camera of an *iPhone 7* and changed the settings to record videos in 60 frames per second (default is 30) which is more than two times higher than the frame rate of the test video itself. The synchronization accuracy is defined as the frame difference between the slowest and fastest players. We used the first display of the video wall as a reference for the measurement. Figure 6.3 shows the maximum frame difference between the slowest and fastest players and the average frame difference for all players. We can see that most of the time the maximum frame difference is only one frame except in the time interval *00:17:48:17 - 00:17:50:00*. This happens because we changed the playback position in the control AAC which requires all players to adjust their position. It took about 2 seconds for all players to buffer the content of the new position and until the synchronization stabilizes again.

There is another strategy which can be applied for seeking, namely pausing the video and waiting for all players to buffer enough data in the new position. After starting the playback from the paused state, a frame-accurate synchronization can be reached immediately as we can see in the chart at time *00:19:48:00* (the video was paused before this time). As a conclusion, the implemented synchronization algorithm delivered a good result with maximal frame difference of only one frame. The ideal

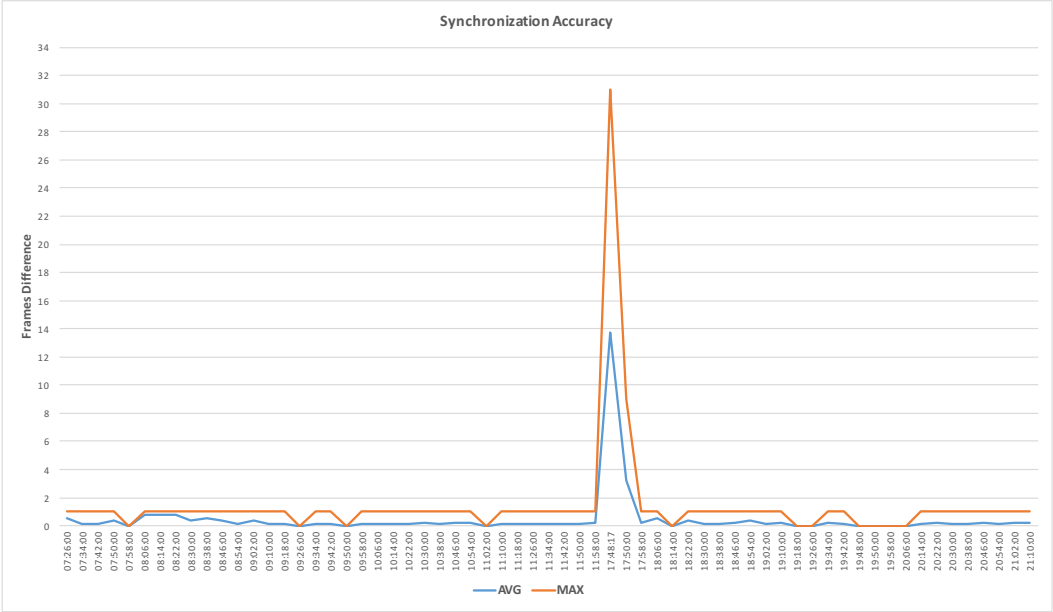


Figure 6.3.: Video Wall Synchronization Accuracy

synchronization result is when all displays show the frame with the same number (frame difference is 0) at any time. This result can be achieved with our algorithm when the synchronization is integrated at a lower level of the browser's media engine, which we could not consider in our implementation without manipulating the browser. The reason behind this lies in the fact that the methods for reading and changing the current video time in JavaScript are not as accurate as if they were implemented natively at the platform level.

6.2 Multiscreen Application Runtime Approaches

In Section 4.4.1 we introduced the three multiscreen application runtime approaches *Multiple Execution Contexts*, *Single Execution Context*, and *Cloud Execution*. This section evaluates the three approaches according to the metrics shown in Table 4.1 which provides only a high-level comparison. For this, we developed two multiscreen applications which are briefly described below:

- **Simple Application**: This application consists of a sender and receiver components. The sender offers a "cast" button that allows launching the receiver component on a target device, and then establish a communication channel between both components. Afterward, the sender reads the system time every *20ms* and displays it in the format *hh:mm:ss.SSS* (SSS are the milliseconds). The time displayed on the sender will also be sent over the established communication channel to the receiver to display it. This method allows us to measure the Photon-To-Motion latency as we will see later.

- **Video Application**: The video application is identical to the first application with the only difference that the receiver component plays the *Big Buck Bunny* video and the time sent by the sender will be displayed on top of it. The selected *Big Buck Bunny* version has a resolution of 1280x720 pixels and a frame rate of 30 frames per second.

There are of course other more complex applications like multiscreen games with extensive graphics processing that could be also used for the evaluation but the results can vary considerably depending on the processing capabilities of the devices under considerations. Our goal is to evaluate the multiscreen approaches according the metrics listed below and not the performance of the application itself on each single device. For complex applications, the evaluation values can be different because they contain not only the values for the evaluation metrics, but also the values for the execution and rendering of the application itself. Therefore, we selected very basic multiscreen applications and evaluated them on the following devices:

- **Sender Device**: We used a *MacBook Pro* with a *3,1 GHz Intel Core i7* CPU with 2 cores, *Intel Iris Graphics 6100 1536 MB* integrated graphics and a memory of *16 GB 1867 MHz DDR3*. It is important to mention that the *CPU*, *Memory*, and *Energy Impact* evaluation results also include the usage of the integrated graphics.
- **Receiver Device**: The receiver device used in the evaluation is *Chromecast Ultra*[1], a widely used low-cost HDMI streaming device from Google. It can be connected to the internet via Wi-Fi or Ethernet and includes a *Marvell Armada 1500 Mini Plus* processor that supports 4K video playback and has 512MB of memory.
- **Cloud Server**: This server is only relevant for the evaluation of the *Cloud Execution* approach and runs on a Microsoft Windows machine with an *Intel Core i7-6820HQ* CPU with 4 cores, integrated *Intel Graphics HD 530*, *16 GB* of memory.

The sender and receiver devices were connected to the same local network, and the bandwidth of the Internet connection was 6Mbps. The evaluation was performed according to the following metrics, which are derived from the non-functional requirements identified in Section 3.2.2.

- **Bitrate**: The bitrate (bandwidth) [Kbps] required by the application during runtime for the communication between the sender, receiver and server (in case of cloud rendering).

[1] https://store.google.com/product/chromecast_ultra

- **Motion-To-Photon Latency**: This is the time [ms] required until a user interaction performed on the sender device is reflected on the receiver device. Both, the Simple and Video multiscreen applications mentioned above display the sender time on the sender and receiver devices. The output of both devices is captured with the iPhone7 camera in slow-motion mode (240 frames per second) which provides high accuracy measurements.
- **CPU Usage**: This metric measures the percentage of processing time used by the multiscreen application on a particular device (Sender or Receiver) or on the server in case the Cloud Rendering approach is considered.
- **Memory Usage**: Memory [MB] used by all processes of the multiscreen application and the underlying runtime on the end-user device or the Server in the case of the Cloud Rendering approach.
- **Energy Impact**: The energy impact is an indication of the power consumption of the application on the sender device provided via the activity monitor on the Mac. It is "*a relative measure of the current energy consumption of the app. Lower numbers are better*" [168]. It takes into account CPU usage, GPU usage, network and disk activities.

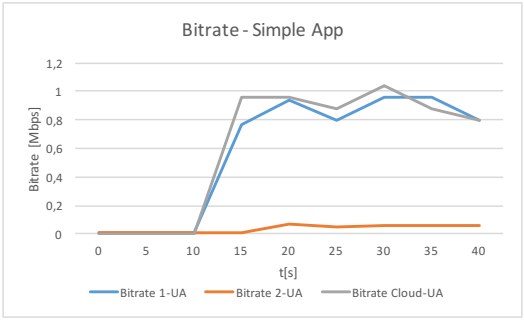
6.2.1 Evaluation of the Simple Application

The evaluation results of the simple application are shown in Figure 6.4. Each chart shows the evaluation results of the three approaches. It is worth to note that the x-axis in all diagrams represents the time of each measurement in seconds. The following list shows the legend of the rendering approaches in all charts:

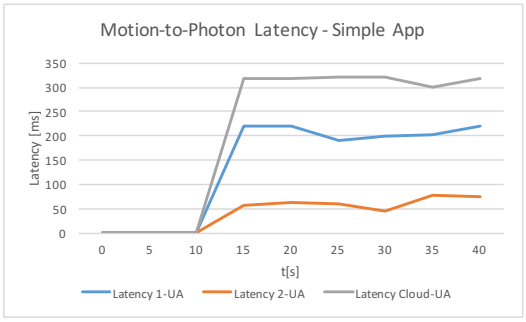
- **1-UA** (Single User Agent): **Single Execution Context**
- **2-UA** (Two User Agents): **Multiple Execution Contexts**
- **Cloud-UA** (Cloud User Agent): **Cloud Execution**

Below is a discussion of the evaluation results:

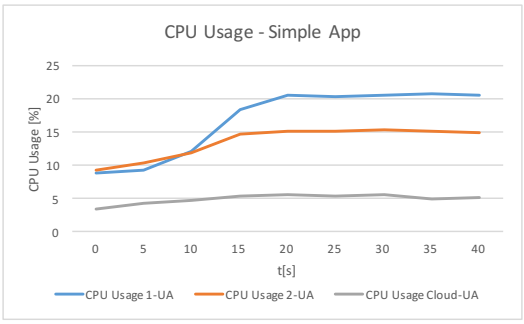
- **Bitrate**: The bitrate usage is shown in Figure 6.4a. As we can see, the bitrate in the 2-UA mode is insignificant compared to the other two approaches. The reason for this is that in the 2-UA mode, the data messages are sent directly to the receiver while in the other two approaches the UI of the receiver application is rendered in headless mode, i.e., the output is captured and streamed as video to the receiver device. The impact of the bitrate is more relevant for the Cloud Execution approach. In this case, the video content is transferred to the client over the Internet where the available bandwidth may be limited compared to the *Single Execution Context* where the video is streamed directly from the sender to the receiver device over the local network.



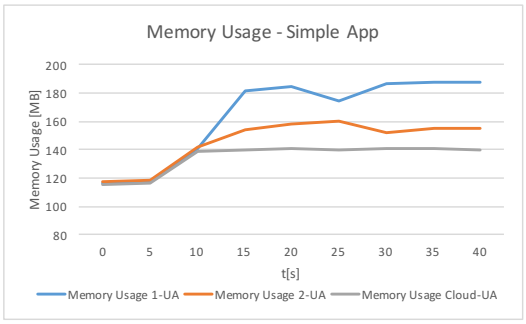
(a) Bitrate



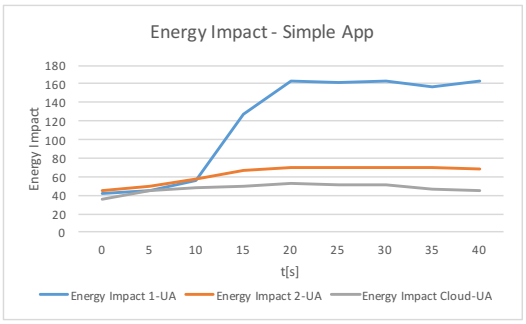
(b) Motion-To-Photon latency



(c) CPU usage



(d) Memory usage



(e) Energy impact

Figure 6.4.: Evaluation of the 3 runtime approaches using a simple application

- **Motion-To-Photon Latency**: Regarding Motion-To-Photon latency, we can see in Figure 6.4b that the 2-UA approach achieves the best result followed by the 1-UA and Cloud-UA approaches. This result is expected since, in the 2-UA approach, the sender transmits the application runtime data (e.g. in JSON format) directly to the receiver in the local network where the transmission latency is negligible. The reason why the Motion-To-Photon latency is around *50ms* on average despite the low transmission latency, is that it also includes the time the receiver UA needs to parse the message and update the application in addition to the time until the changes are reflected on the display. This shows why a low-cost streaming device like Chromecast still has a certain latency that needs to be considered. For example, the Motion-To-Photon latency will be lower when a high-performance device such as a game console

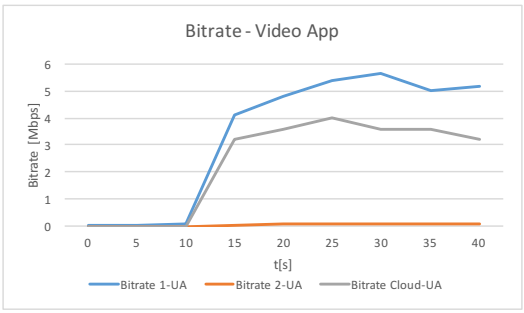
is used as the receiver. The reason why the Motion-To-Photon latency is higher for the 1-UA and Cloud UA approaches than for the 2-UA approach is that additional video encoding (sender side) and decoding (receiver side) steps are required. This requires buffering some amount of video data in order to react to network fluctuations especially in the Cloud-UA approach where the video is transmitted over the Internet to the receiver.

- **CPU Usage**: The evaluation of the CPU usage is shown in Figure 6.4c. We can see that the usage of the Cloud-UA approach is the lowest compared to the other two approaches since the sender only needs to play a video without any application processing. The 2-UA approach is second because the CPU utilization involves the execution of the sender application and the transfer of data to the receiver. The highest CPU usage is measured for the 1-UA approach since both applications (sender and receiver) are executed on the sender device (receiver application in headless mode), and the receiver application UI will be captured and transmitted to the receiver device as a video stream.
- **Memory Usage**: The evaluation results of the memory usage which are shown in Figure 6.4a are similar to the CPU usage results. The Cloud-UA approach requires memory as a buffer for decoding the video which has a low bitrate (around 1Mbps) for the simple application. On the second place there is the 2-UA approach which requires a certain amount of memory for executing the sender application and for the underlying application runtime. Finally, the 1-UA approach requires the highest amount of memory for executing both applications and for encoding a video from the headless receiver application.
- **Energy Impact**: Finally, the energy impact evaluation shown in Figure 6.4e shows that the energy consumption on the sender device is the highest in the 1-UA approach and lowest in the Cloud-UA approach. As expected, the 1-UA consumes more energy than the other two approaches since it executes two applications and encodes and streams a video. Regarding the 2-UA and Cloud-UA approaches, the results show that the energy consumption for decoding a low bitrate video for the simple application is lower than the energy consumption for executing the application itself.

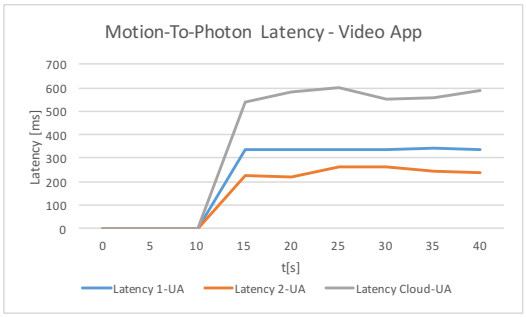
From the evaluation results, it is clear that the *Multiple Execution Contexts* (*2-UA*) approach is the better choice if the receiving device has enough power to run the application without affecting the use experience. For demanding applications such as games that cannot be processed on the device under consideration due to lack of resources, the *Cloud Execution* approach can be used at the costs of higher bandwidth consumption and the costs of operating and maintaining a cloud runtime environment. Cloud gaming platforms such as Google Stadia [134] use this approach to enable gaming applications on low-performance devices such as Chromecast.

6.2.2 Evaluation of the Video Application

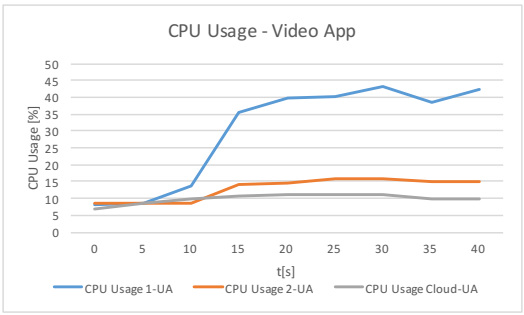
The evaluation results of the video application are shown in Figure 6.5. The structure is the same as above:



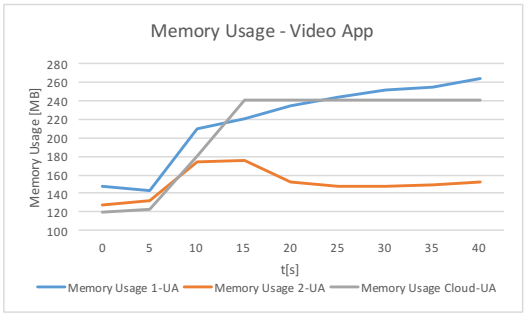
(a) Bitrate



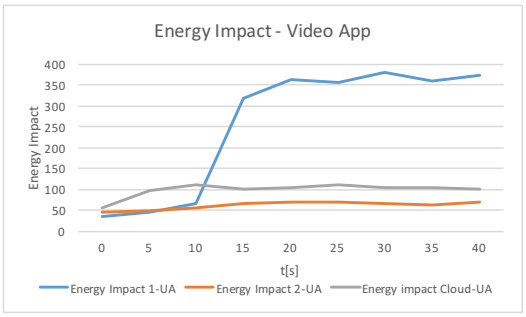
(b) Motion-To-Photon latency



(c) CPU usage



(d) Memory usage



(e) Energy impact

Figure 6.5.: Evaluation of the 3 runtime approaches using a video application

- **Bitrate**: Figure 6.5a shows a similar distribution for the bitrate as for the simple application with the difference, that the 1-UA and Cloud-UA approach that capture and stream the receiver application as video require more bandwidth compared to the simple application. We can also see that the bitrate of the Cloud-UA approach is around 30% lower than for the 1-UA approach due to the compression settings in the video encoder on the server in order to provide

a smooth playback on the receiver, in case the video is streamed over the Internet. The bitrate can vary when network conditions change.

- **Motion-To-Photon Latency**: Regarding Motion-To-Photon latency, we can also see that the ranking of the three approaches is the same as in the simple application but with higher latency up to 600ms on average for the Cloud-UA approach which is an expected result due to the higher video bitrate and video encoding or decoding times (Figure 6.5b). On the other hand, we expected that the latency for the 2-UA approach remains the same as in the simple application. However Figure 6.5b shows that the latency increased from 50ms to 250ms on average. The explanation is that the receiver is a low-performance device and needs more time to display the received data from the sender if it plays a video at the same time.
- **CPU Usage**: The results of the CPU usage evaluation are shown in Figure 6.5c. There is no difference in the 2-UA approach between the evaluation of the simple and video applications which is expected since the sender component is the same in both applications. However, we can see that there is an increase in the CPU usage for the 1-UA and Cloud-UA approaches compared to the simple application. The explanation for this increase is the additional processing resources needed to decode and play a high bitrate video in the video application.
- **Memory Usage**: As for the CPU usage, the memory usage shown in Figure 6.5a for the simple and video applications is the same for the 2-UA approach and increases in the other two approaches due to the higher video bitrates.
- **Energy Impact**: The energy impact evaluation is shown in Figure 6.5e. Also, the energy impact for the simple and video applications is the same for the 2-UA approach and increases in the other two approaches due to the higher video bitrates.

The evaluation of the video application shows similar results as for the simple application. Here is also the *Multiple Execution Contexts* (*2-UA*) approach is the better choice if the end device is able to play the video and supports the corresponding codecs. In case the end device cannot render the video locally, for example 360° videos which require additional processing resources compared to normal videos to perform the geometrical transformation, then the *Cloud Execution* approach can be applied. The evaluation of 360° video streaming and playback will be discussed in Section in more detail.

6.2.3 Evaluation of the Cloud-UA Approach on the Server

So far, we evaluated and analyzed the three multiscreen runtime approaches 1-UA, 2-UA, and Cloud-UA using a simple and a video application. In the Cloud-UA approach,

it is also important to evaluate the resources (CPU and memory) used on the server for both applications. These evaluation results are shown in Figure 6.6.

(a) CPU Usage **(b)** Memory Usage

Figure 6.6.: Evaluation of server resources for the Cloud-UA approach

- **CPU Usage**: Figure 6.6a shows that the CPU usage for the video application is higher than for the simple application which is an expected result because the applications were started on the server at time 0 while the capturing and streaming at time 15 which explains the increase in the CPU usage at this time.
- **Memory Usage**: Figure 6.6b shows that the memory usage for the video application is higher than for the simple application which is related to the amount of video data that needs to be buffered in the memory. The increase in memory consumption for both applications occurs after the capturing and streaming were started at time 15s. This result is also expected since the capturing requires additional memory for the encoding and buffering of the output video.

6.2.4 Summary

We can see from the evaluation of the three approaches that the 2-UA approach provides the best results regarding all metrics. This is because the application execution is distributed on multiple devices and there is no UI capturing and streaming of application components. The question that arises from this result is, why we still need the other two approaches. The answer for this question is that in the 1-UA approach there are no requirments for the application runtime on the receiver device and the sender only needs to support wireless display standards such as Miracast and Airplay which are supported on Android and iOS platforms as well as on the majority of TV platforms such as Tizen, WebOS, and AppleTV. Also from a security & privacy perspective, the 1-UA approach keeps all application data in one place on the sender device and no information will be shared with other devices. Regarding the 2-UA approach, a widely deployed open standard is still missing, but this may change in the near future when the work of the Open Screen Protocol [16] is finished which

is currently developed in the W3C Second Screen Community Group as an open standard. Currently, the most widely deployed solution for the 2-UA approach is the Google Cast Framework supported in Chrome browser (as sender) on all desktop and mobile platforms and receiver devices (Chromecast and Android TV).
Finally, the results of the Cloud-UA approach show that this option is only relevant for high-performance applications that require upscale graphic computation capabilities like games or VR applications. Therefore, the scalability and additional server costs must be weighed against the benefits of this approach. Another use case for this approach is the virtualization of TV applications (which usually run on dedicated hardware such as Set-Top-Boxes) using edge computing paradigms. The W3C Cloud Browser Task Force [133] discusses first ideas for standardizing this approach, but there is still little support from the industry side.

6.3 360° Video Rendering and Streaming

This section evaluates the 360° pre-rendering solution we introduced in Section 5.3.4 and compares it to the CST (Client Side Transformation) and SST (Server Side Transformation) approaches. The three solutions will be evaluated according to the metrics: bitrate usage, client resource usage (includes CPU usage, memory usage, and energy impact), motion-to-photon latency, and used server resources.

6.3.1 Bitrate Usage

To compare the required bandwidth for each of the three approaches, we will use 360° equirectangular videos with 4K (3840x1920) resolution and corresponding FOV videos (60° vertical FOV angle) with HD (1280x720) resolution. More specifically, we will use 8 different 360° videos provided by several German broadcasters like Arte, ZDF, RBB, and BR. All videos are encoded in H.264 and have a frame rate of 30 frames per second. for each of these videos we generated the corresponding FOVs also using the H.264 codec with a GOP size of 10 frames using the pre-rendered approach and then calculated the average FOV bitrate for each video. Figure 6.7 compares the bitrates of the source 360° videos (in blue) which are equivalent to the required bandwidth for the CST approach and the average bitrates of FOV videos (in orange) which are equivalent to the required bandwidth for the SST and pre-rendered approaches. The bitrate overhead for the CST approach compared to the other two approaches is around 83,5% (red line, top area). The comparison between the bitrates of 4K and HD H.264 encoded videos in Figure 5.9 shows an overhead of around 89,3% which is higher than the result of this experiment. The reason for this difference is because the GOP size of the generated FOV videos is set to 10 frames which impacts the compression rate of the encoder. The reason why the GOP

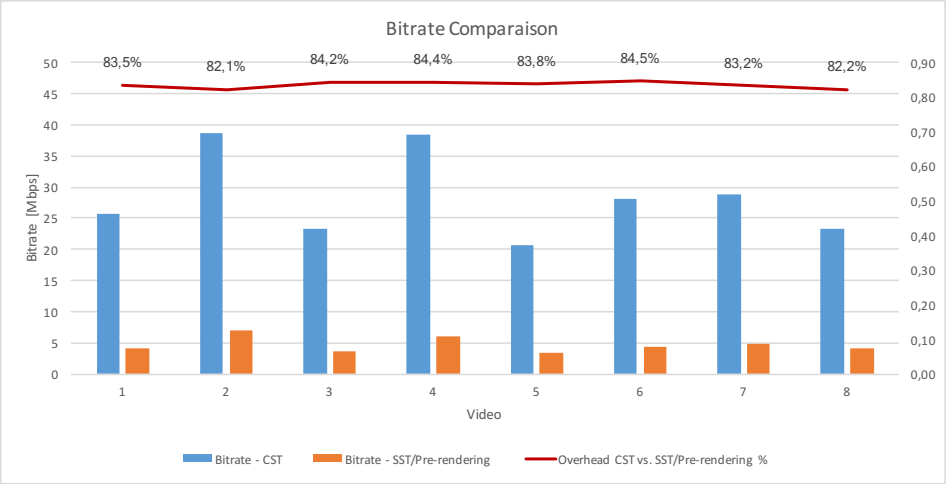


Figure 6.7.: Bitrate overhead for CSP compared to SSP and pre-rendering approaches

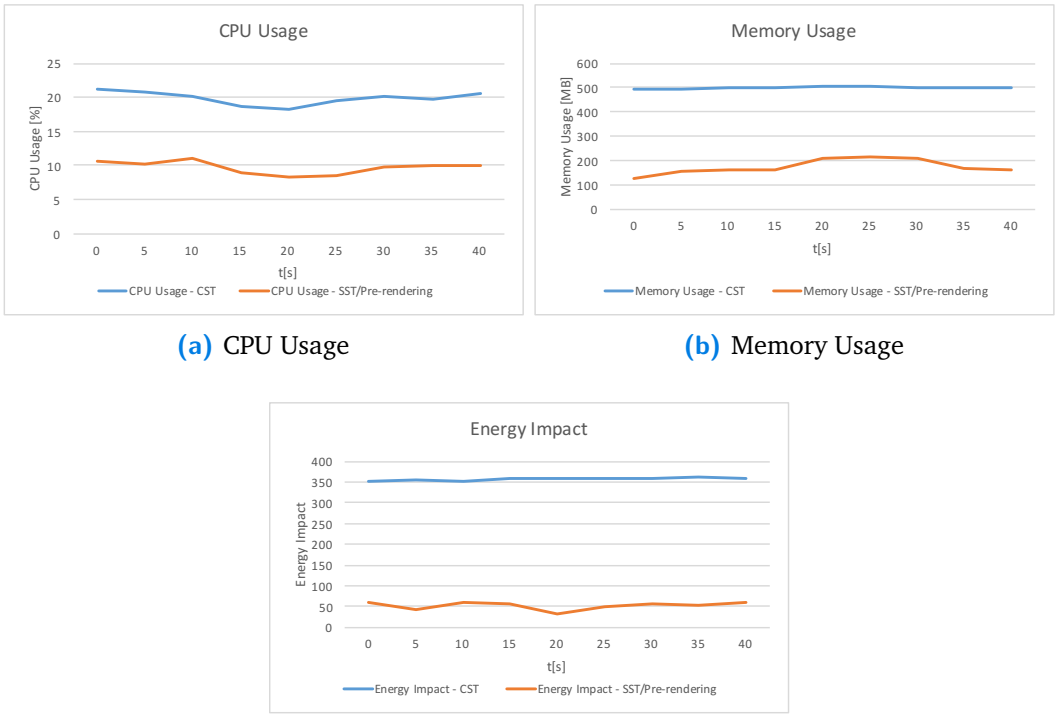
size is set to 10 frames will be explained in the discussion of the motion-to-photon evaluation below.

6.3.2 Client Resources

This usage of client resources for the three 360° rendering and streaming approaches CST, SST, and pre-rendering are shown in Figure 6.8. As we can see from these results, there is no difference between SST and pre-rendering since the client is the same for both approaches and needs only to play the FOV video stream already processed on the server. The client device used in the evaluation is a *MacBook Pro* with a *3,1 GHz Intel Core i7* CPU (2 cores), integrated *Intel Iris Graphics 6100 1536 MB* and *16 GB 1867 MHz DDR3* memory. The content used in the evaluation is a 4K 360° H.264 video with a bitrate of around 30Mbps (used in CST) which is the average bitrate required to encode 4K video in H.264 and FOV bitrate of around 4Mbps (used in SST and pre-rendering) which is the average bitrate required to encode an HD video in H.264. As we can see from these results, the SST and pre-rendering approaches outperform the CST approach regarding all three metrics (*CPU usage*, *memory usage*, and *energy impact*). The CST approach requires 50% more CPU, 65% more memory and consumes 7 times more energy than the SST and pre-rendering approaches. This result is expected since the CST client needs to decode a 4K 360° video and calculate the FOV on the client device while the other two approaches only need to play an HD video without any additional processing.

6.3.3 Motion-To-Photon Latency

The Motion-To-Photon latency (see Section 3.2.2) is one of the essential metrics with a direct impact on the usability of 360° video playback on a specific device. The most



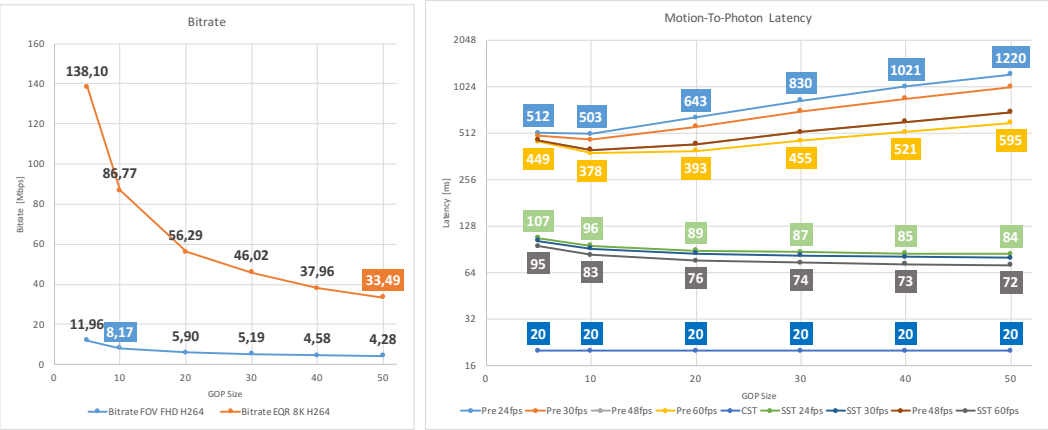
(a) CPU Usage

(b) Memory Usage

(c) Energy Impact

Figure 6.8.: Evaluation of client resources for the three approaches

relevant device categories that are used to display 360° content are head-mounted displays which use motion sensors, mobile devices like smartphones and tablets which use touch inputs, desktop PCs and laptops which use mouse or keyboard inputs, and TV devices that use remote controls as input devices. The evaluation of the Motion-To-Photon latency for the three 360° approaches CST, SST, and pre-rendering is shown in Figure 6.9b but before we analyze these results, we will discuss the impact of GOP size on the bitrate and Motion-To-Photon latency at same time (Figure 6.9a). The figure shows the bitrate of Caminandes 360° Equirectangular video in 8K resolution (8192x4096) with an average bitrate of the generated FOV videos (60°x36°) in FHD resolution (1920x1080) when varying the GOP size. The Peak Signal-to-Noise Ratio (PSNR) which measures the quality of reconstruction of lossy compression codecs (like the H.264 video codec in this evaluation) is kept constant (around 45 dB) in all measurements. We can see that long GOPs provide better compression rates for both videos (8K and FHD). However, large GOPs increase the complexity and thus the required resources for encoding, decoding and even seeking in the video. The preferred GOP size depends on the content and is most often under 50 frames. YouTube recommends a GOP of half of the frame rate for H.264 videos [155]. The GOP size also has an impact on the Motion-To-Photon latency, but only for the pre-rendered approach. The duration of a GOP can be calculated as $D = GOP/FPS$. For example, the duration of a GOP with 10 frames in a video with a frame rate of 25 FPS is 400ms. If the player is at time t in the current GOP and the user makes an interaction to change the FOV, then the player



(a) Impact of GOP size on bitrate **(b)** Motion-To-Photon latency comparison

Figure 6.9.: Motion-To-Photon Latency of 360° Streaming and Rendering Approaches

must continue the current GOP before starting the playback of the new GOP. In this case, the player needs the time $D - t$ to fetch the new GOP (on average $D/2$). The total Motion-To-Motion latency for the pre-rendered approach also includes the time needed to capture the user input, the network latency and the download time for requesting the new GOP, and finally the time to decode and display the video. Since the download time depends on the available bandwidth, we assumed in the experiment that the available bandwidth is equal to the bitrate of the source 360° 8K video in order to compare the different approaches fairly. Since the CST approach is independent of the GOP size of the source video, we selected a bitrate of 33,49 Mbps which corresponds to a GOP size of 50 frames. In this case, the duration of a GOP in a video with a frame rate of 25 FPS is 2s, which is a widely used segment length for adaptive streaming. If we consider a GOP size of 10 frames, we can see that the bitrate of the FOV video is 8,17Mbps. The Motion-To-Photon latency in Figure 6.9b shows the highest values for the pre-rendered approach followed by the SST approach and then the CST approach. For example, the Motion-To-Photon latency for a GOP size of 10 frames and a video frame rate of 24 FPS is around 503ms for the pre-rendered approach, 96ms for the SST approach and 20ms for the CST approach. Based on these results, we can see that only the CST approach is suitable for HMDs, since a latency of more than 20ms leads to motion sickness. The pre-rendered approach is suitable for devices that use keyboard or remote control as input. This is because the user is not interacting directly with the content itself (e.g., via dragging on a touch screen to change the FOV, where the user expects the video content of the touched video area to remain under his finger), but using a second device like the TV remote control to change the view on the TV. TV viewers are used to experience delays when they interact with video services, e.g. when switching between channels. The pre-rendered approach has already been used successfully with the broadcasters WDR, ZDF (Germany) and ERT (Greece) as VOD and live streams on HbbTV-enabled terminals. For example, the Biathlon World Cup 2019 in

Oberhof/Germany was available as a 360° live stream in the HbbTV application of the German public broadcaster ZDF using our solution [169]. It was very exciting to see that during the Biathlon World Cup the number of viewers who watched the 360° live stream in HbbTV was very high and almost as high as the number of viewers who watched the 360° live stream in the VR app for mobile devices and HMDs. This shows that the pre-rendered approach is a good choice for TV sets.

6.3.4 Server Resources

Storage: The CST and SST approaches operate directly on the source 360° video while the pre-rendered approach creates N different FOV videos where N depends on $\Delta\phi$ and $\Delta\theta$ as explained in Section 5.3.4. Our experience has shown that $\Delta\phi = 30°$ and $\Delta\theta = 60°$ provide a good user experience on TV for most videos when using the remote control as input device. The total number N of FOV videos will be in this case 180. The bitrate of a FOV video is 16,5% of the source 360° video (see Section 6.3.1) on average. This means that the total bitrate of all FOV videos is $180 * 0,165 = 29,7$ times higher than the bitrate of the 360° video. In other words, the storage required for the pre-rendering approach is around 30 times higher than the storage required for the other two approaches.

Rendering: In the CST approach, the rendering happens in the client without involving any server. In the SST approach, a server instance is needed for each session to render the video in the cloud. In our experiment, we selected Amazon AWS EC2 instances equipped with the new generation of NVIDIA GPUs for the SST and pre-rendering approaches. We used the smallest GPU-based EC2 instance type offered by AWS which is fully capable of rendering 360° videos up to resolution of 4K in real time. Other more powerful GPU-based EC2 instance types can also be used, but it is an overhead to use them for 360° video rendering in 4K resolution. According to Amazon pricing, each instance of this type costs around 1,14$/h for the US East region. This is also the costs for each SST session per hour. Regarding the pre-rendering approach, a server instance is used only for generating the FOV videos which are made available to clients via CDNs.

6.3.5 Summary

From the evaluation results of the three 360° approaches, we can see that each of these approaches has its advantages and disadvantages. CST is the only approach that can be applied to HMDs due to the Motion-To-Photon latency requirement of under 20ms. This can be achieved, if there is enough bandwidth to deliver the 360° video in real time and the are sufficient graphical processing resources (GPU) on

the client to perform the 360° transformation which is not available on embedded devices such as TV sets. If at least one of these two requirements is not fulfilled, SST and pre-rendering can be used. The SST approach can be applied to all device types except HMDs. However, it is costly and does not scale for massive 360° video delivery since each client (360° player) requires a GPU server instance running in the cloud or on the edge to render and stream the 360° video. But this approach is gaining a lot of attraction in the gaming industry where costumers are ready to pay for such a service to play games on any device even on low capability devices like TVs. For example, Google recently announced the launch of the new cloud gaming platform Stadia [134], which is able to stream games up to a resolution of 4K on almost any screen, including low capability devices like Chromecast. "Stadia works across various connections from 35 Mbps down to a recommended minimum of 10 Mbps" [170]. The pre-rendering solution introduced in this thesis solves the scalability issue concerning the required graphical processing resources of the SST approach and the bandwidth and processing issues of the CST approach at the cost of increasing the Motion-To-Photon latency. Our approach is applicable to TVs that use remote controls or arrow keys for navigation with acceptable user experience. This has been proven by the use of our approach on various broadcasters in HbbTV, as mentioned above.

7 Conclusions and Outlook

7.1 Conclusions

In this thesis, new concepts for modelling and developing multiscreen applications as well as a new approach for the creation, delivery, and playback of multimedia content in a multiscreen environment with a focus on 360° videos were presented. Key multimedia multiscreen use cases and application scenarios have been considered to derive the requirements of the application model and the underlying framework. The research questions identified in Section 1.2 were addressed in the following ways.

***Research Question 1**: How to design and develop multiscreen applications, taking into account aspects such as development costs and time, platform coverage and interoperability between devices and technology silos.*

This research question was addressed in this thesis from two different viewpoints: The conceptual design of multiscreen applications was analyzed independently of the underlying framework and utilized technologies. This enables the modelling of multiscreen applications without being dependent on the underlying platform. Once the concepts and models of a multiscreen application have been created, it can be mapped to technologies supported by the platforms on the devices under consideration. This thesis investigated this aspect since there are no comparable methods and tools for modelling and designing multiscreen applications, while there are already well-proven concepts and design patterns for single-screen applications such as the Model View Controller (MVC) paradigm [171]. More specifically, in Section 4.2 of this thesis, a new method called *Multiscreen Model Tree* was presented that allows the modelling of a multiscreen application and its components in every phase of its lifecycle. The newly introduced method supports the core multiscreen functions identified in Section 3.2 such as discovery, launch, joining, instantiation, mirroring, and migration of application components. The fundamental elements of the multiscreen model tree are the application components which can be either composite or atomic. This classification enables the reusability of the components, especially the atomic ones, and the capability to migrate, instantiate or mirror them across heterogeneous devices at any time and without additional effort for the developer.

This approach reduces development costs and times on one hand, and the maintainability and expandability of the application on the other hand. It also enables the distribution of application components to devices with heterogeneous platforms without having to reimplement the entire application, but only individual components for the desired platforms. This provides an increased "Separation of Concerns" according to the modern software engineering principles. Concerning the interaction among the application components during runtime, this thesis has identified the three well-suited approaches *Message-Driven*, *Event-Driven*, and *Data-Driven* described and considered the realization of each approach in centralized and decentralized environments. Developers of multiscreen applications can select the appropriate approach that fits the application scenario based on given criteria and requirements. We showed that for complex applications with distributed logic and where components can be migrated among devices, it is beneficial to use the *Data-Driven* approach since the state of any component is preserved after migration and new instances can access the current state without additional application logic. The *Event-Driven* and *Message-Driven* approaches are recommended for applications where it is not necessary to share the state between components. The second part of this research question is about the platform coverage and interoperability between devices. This thesis introduced a concept for using Web technologies and especially *Web Components* to support the proposed multiscreen application model as the Web has quickly developed towards a platform for multimedia applications across multiple devices and platforms.

***Research Question 2**: How to efficiently distribute and run multiscreen applications, taking into account available resources such as bandwidth, processing, storage and battery without affecting the user experience.*

This thesis has identified the three approaches *Single-Execution Context*, *Multiple-Execution Contexts* and *Cloud-Execution* for the multiscreen runtime. All support the multiscreen application model we discussed in the first research question. It is worth mentioning that it is not necessary to modify the multiscreen application in order to support one of the three runtime approaches. These approaches have been evaluated according to multiple metrics listed in this research question. The results have shown that the *Multiple-Execution Contexts* is the preferable approach and outperforms the other two approaches regarding all metrics. We still need to consider the other two approaches: The *Single-Execution Context* must be used if the target device does not provide an application runtime environment, but only video playback capabilities. Therefore, the target application needs to be executed in "headless mode" on the host device, and the user interface will be captured and sent to the target device. Besides the high processing and battery consumption, this approach is limited to two devices. The *Cloud-Execution* approach is similar but offloads the application runtime to a server running in the cloud and only sends the video stream of each

application component to the corresponding device. This approach is relevant for specific use cases like gaming and VR or AR applications in case client devices are not able to perform the complex graphics processing locally. The main limitation of this approach is the hard limit on the Motion-to-Photon latency of 20ms which is difficult to achieve in current networks. 5G could enable this kind of use cases in the future but is not available yet. Another problem with this approach is the scalability of using server graphics processing resources to deliver 360° videos to the mass audience and the resulting high operating costs.

***Research Question 3**: How to efficiently prepare, stream and play multimedia content, especially 360° videos, across different platforms taking into account available bandwidth, content quality, media rendering capabilities and available resources on target devices.*

This research question addresses multimedia content in a multiscreen environment. There are already existing solutions for adaptive streaming and playback of multimedia content across different devices and platforms such as MPEG-DASH and HLS which are essential for any multiscreen multimedia application. For example, if an atomic component that plays a video is migrated from one device to another, the media playback will adapt automatically to the target device, i.e., by selecting the stream with the appropriate video and audio codec, resolution, and bitrate. In contrast, this thesis focused on the open research questions of sharing and synchronization of adaptive media content across devices. For this, the multiscreen application framework was extended with an API which allows to play and control media content on remote devices with the ability to synchronize media streams across devices. The developed approach makes it easy to synchronize videos across multiple devices just by adding the video elements under consideration into the same *sync group* with just a single line of code. The synchronization algorithm presented in this thesis was implemented and evaluated as proof-of-concept.

Another focus of this research question is the preparation, delivery, and playback of 360° videos in a multiscreen environment. Most state-of-the-art contributions in the domain of delivery and playback of 360° videos are focused on HMDs. In a multiscreen environment, it is important also to consider other device categories like TVs, for example, to allow broadcasters to deliver 360° videos to the same device used for traditional channels like HbbTV. However, HbbTV does not offer the APIs needed for rendering the 360° locally, and even most modern TVs are not capable of rendering 360° videos due to limited processing resources. To remedy this situation, we introduced a novel mechanism for the playback of high-quality 360° videos on low-capability devices based on the pre-rendering of multiple FOV combinations. The main advantage of this approach is that it does not require any processing resources neither on the server nor on the client after the content is generated and

made available through a CDN. The evaluation of our approach compared to the two main state-of-the-art approaches CST and SST confirms the benefits of our approach regarding processing requirements, scalability, bandwidth, and content quality. One limitation is the high Motion-To-Photon latency, so this approach is limited to devices that support navigation using arrow keys like TV remote controls but is not suitable for HMDs.

***Research Question 4**: How to support the standardization of an interoperable and flexible model for distributed multiscreen applications and the specification of related standard APIs and network protocols.*

Interoperability is a key requirement for any multiscreen solution since multiscreen applications can be distributed on devices by different manufacturers and running different platforms. In this work, we considered this aspect on three different levels: First, the application runtime which runs application components developed using technologies supported by the underlying platforms. In this thesis, we focused on Web standards which offer open technologies to develop interoperable rich multimedia applications. Since the multiscreen application model introduced in this thesis is independent of the underlying runtime environment, other technologies can be considered in a similar way, but these were not in the focus of this work.

The second level of interoperability is a set of standard Web APIs that support key multiscreen features like discovery, launch, joining, communication, synchronization, and remote playback from the Web runtime by taking security and privacy aspects into account. These APIs are being developed in the W3C Second Screen Working Group [12]. The author of this thesis is a member of this standardization effort since 2013 and is an active contributor to the *Presentation API* [13] and *Remote Playback API* [14] which are both *Candidate Recommendations* of the W3C. The author of this thesis also earned the role of *test facilitator* to ensure the compatibility of implementations with the API specifications. The contributions of the author to the specifications is influenced by the requirements identified and results achieved in this thesis.

The third level of interoperability is the network protocol layer. Without these protocols, it is difficult to achieve interoperability across different vendors. For example, the current implementations of both APIs in the Chrome browser are built on top of the proprietary Google Cast Protocol [9]. Therefore, the work on a new protocol called *Open Screen Protocol* [16] is started in the Second Screen Community Group [15] to solve this issue. It is expected that a first draft of the protocol will be published in 2020. Several results of this thesis have contributed to the community group, especially a proposal to support non-web environments in the protocol.

7.2 Outlook

There are several opportunities for expanding the outcomes of this thesis. First, it may be worth to investigate the applicability of the multiscreen application model introduced in this work in a non-web environment and to provide a proof-of-concept implementation for a specific platform. This can be achieved by using the *Open Screen Protocol* as a foundation for the implementation. Therefore, the priority for future activities is to continue contributing the results of this work to the W3C Second Screen Community Group to accelerate the development of the Open Screen Protocol and also to consider its integration with other standards such as HbbTV.
Another outcome of this work, which is also worth further investigation, is the pre-rendered approach for 360° video streaming and playback. There are different directions for expanding this research activity: 1) reduce the Motion-to-Photon latency to support more devices like smartphones and tablets, 2) investigate new algorithms for the transition between FOV videos based on the bitrates of the different representations and 3) introducing new features like the pre-rendering of transition videos along paths that connect points of interest in a 360° video.

Bibliography

[1] Google. *The New Multi-Screen World Study*. Research Study. Online: https://www.thinkwithgoogle.com/advertising-channels/mobile/the-new-multi-screen-world-study/. Google, June 2012 (cit. on p. 1).

[2] *Netflix Supported Devices*. Electronic Document. Online: https://devices.netflix.com/ (cit. on p. 1).

[3] Cisco. *Cisco Visual Networking Index: Forecast and Methodology, 2016–2021*. White paper. Online: http://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf. Cisco, June 2017 (cit. on p. 1).

[4] *YouTube*. Electronic Document. Online: https://www.youtube.com (cit. on pp. 2, 9, 41).

[5] *Facebook*. Electronic Document. Online: http://www.facebook.com (cit. on pp. 2, 9).

[6] *Airplay*. Electronic Document. Online: https://developer.apple.com/airplay/ (cit. on pp. 2, 14, 81, 88, 115).

[7] *Apple TV*. Electronic Document. Online: https://www.apple.com/tv/ (cit. on pp. 2, 9).

[8] *Miracast - High-definition content sharing on Wi-Fi devices everywhere*. Electronic Document. Online: https://www.wi-fi.org/discover-wi-fi/miracast (cit. on pp. 2, 15, 81, 88, 115).

[9] *Google Cast*. Electronic Document. Online: https://developers.google.com/cast/ (cit. on pp. 2, 80, 172).

[10] *Chromecast*. Electronic Document. Online: https://google.com/chromecast (cit. on pp. 2, 9).

[11] *World Wide Web Consortium (W3C)*. Electronic Document. Online: https://www.w3.org (cit. on pp. 4, 56).

[12] W3C. *Second Screen Working Group*. Tech. rep. Online: https://www.w3.org/2014/secondscreen/. The World Wide Web Consortium (W3C), 2017 (cit. on pp. 4, 22, 87, 172, 204).

[13] *Presentation API, Candidate Recommendation*. Technical Report. Online: https://www.w3.org/TR/presentation-api/. The World Wide Web Consortium (W3C), 2017 (cit. on pp. 4, 22, 87, 172).

[14]*Remote Playback API, Candidate Recommendation*. Technical Report. Online: https://www.w3.org/TR/remote-playback/. The World Wide Web Consortium (W3C), 2017 (cit. on pp. 4, 22, 87, 172).

[15]W3C. *Second Screen Community Group*. Tech. rep. Online: https://www.w3.org/community/webscreens/. The World Wide Web Consortium (W3C), 2017 (cit. on pp. 4, 172).

[16]*Open Screen Protocol*. Open Source Specification. Online: https://github.com/webscreens/openscreenprotocol. The World Wide Web Consortium (W3C), 2017 (cit. on pp. 4, 87, 161, 172, 204).

[17]Louay Bassbouss, Max Tritschler, Stephan Steglich, Kiyoshi Tanaka, and Yasuhiko Miyazaki. „Towards a Multi-screen Application Model for the Web“. In: *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*. Kyoto, Japan, 2013, pp. 528–533 (cit. on pp. 4, 22, 90, 91, 203).

[18]Louay Bassbouss, Görkem Güçlü, and Stephan Steglich. „Towards a wake-up and synchronization mechanism for Multiscreen applications using iBeacon“. In: *2014 International Conference on Signal Processing and Multimedia Applications (SIGMAP)*. Vienna, Austria, 2014, pp. 67–72 (cit. on pp. 4, 104, 108, 202).

[19]Louay Bassbouss, Stephan Steglich, and Martin Lasak. „Best Paper Award: High Quality 360° Video Rendering and Streaming“. In: *Media and ICT for the Creative Industries*. Porto, Portugal, 2016 (cit. on pp. 5, 127, 202).

[20]Louay Bassbouss, Stephan Steglich, and Sascha Braun. „Towards a high efficient 360° video processing and streaming solution in a multiscreen environment“. In: *2017 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*. 2017, pp. 417–422 (cit. on pp. 5, 127, 201).

[21]Louay Bassbouss, Stefan Pham, and Stephan Steglich. „Streaming and Playback of 16K 360° Videos on the Web“. In: *2018 IEEE Middle East and North Africa Communications Conference (MENACOMM) (IEEE MENACOMM'18)*. Jounieh, Lebanon, 2018 (cit. on pp. 5, 127, 133, 147, 201).

[22]Leon Cruickshank, Emmanuel Tsekleves, Roger Whitham, Annette Hill, and Kaoruko Kondo. „Making interactive TV easier to use: Interface design for a second screen approach“. In: *The Design Journal* 10.3 (2007), pp. 41–53 (cit. on p. 7).

[23]M. Mu, W. Knowles, Y. Sani, A. Mauthe, and N. Race. „Improving Interactive TV Experience Using Second Screen Mobile Applications“. In: *2015 IEEE International Symposium on Multimedia (ISM)*. 2015, pp. 373–376 (cit. on p. 8).

[24]*Netflix*. Electronic Document. Online: https://www.netflix.com (cit. on pp. 9, 11, 41).

[25]*Netflix Hack Day - Spring 2016*. Electronic Document. Online: http://techblog.netflix.com/2016/05/netflix-hack-day-spring-2016.html (cit. on p. 9).

[26]*Google Slides*. Electronic Document. Online: https://www.google.com/slides/about/ (cit. on p. 9).

[27]James Blake. „Second screen interaction in the cinema: Experimenting with transmedia narratives and commercializing user participation“. In: *Participations Journal if Audience and Reception Studies* 14 (2017) (cit. on p. 9).

[28]Florian Pfeffel, Peter Kexel, Christoph A. Kexel, and Ratz Maria. „Second Screen: User Behaviour of Spectators while Watching Football“. In: *Athens Journal of Sports*. Online: https://www.athensjournals.gr/sports/2016-3-2-2-Pfeffel.pdf. June 2016, pp. 119–128 (cit. on p. 9).

[29]„How synchronizing TV and online ads helped Nissan to boost brand awareness“. In: *White Paper: Multi-Screen Study – Nissan* (Apr. 2015) (cit. on p. 9).

[30]*Shazam - Music Discovery, Charts & Song Lyrics*. Electronic Document. Online: https://www.shazam.com/ (cit. on p. 9).

[31]*The Walking Dead - Story Sync - AMC*. Electronic Document. Online: http://www.amc.com/shows/the-walking-dead/story-sync/ (cit. on p. 9).

[32]*360 Videos | Virtual Reality im ZDF*. Electronic Document. Online: http://vr.zdf.de/ (cit. on p. 9).

[33]*Arte360 VR*. Electronic Document. Online: https://sites.arte.tv/360/en (cit. on p. 9).

[34]*Red Bull VR Hub*. Electronic Document. Online: https://www.redbull.com/vr (cit. on p. 9).

[35]*Virtual Reality - YouTube*. Electronic Document. Online: https://www.youtube.com/vr (cit. on p. 10).

[36]Andrew Donoho, Bryan Roe, Maarten Bodlaender, et al. *UPnP Device Architecture 2.0*. Electronic Document. Online: http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v2.0.pdf. 2015 (cit. on pp. 10, 87).

[37]J. Postel. *UPnP Device Architecture 2.0*. Electronic Document. Online: http://tools.ietf.org/html/rfc768. 1980 (cit. on p. 10).

[38]UPnP Forum. *UPnP Standards & Architecture*. Electronic Document. Online: http://upnp.org (cit. on pp. 10, 24, 51).

[39]„DIAL - Discovery and Launch protocol specification 2.1“. In: (Sept. 2017). Online: http://www.dial-multiscreen.org/dial-protocol-specification (cit. on pp. 11, 80, 88, 114).

[40]S. Cheshire and M. Krochmal. „Multicast DNS“. In: (Feb. 2013). Online: http://tools.ietf.org/html/rfc6762 (cit. on pp. 11, 24, 87).

[41]S. Cheshire and M. Krochmal. „DNS-Based Service Discovery“. In: (Feb. 2013). Online: http://tools.ietf.org/html/rfc6763 (cit. on p. 11).

[42]*HbbTV 2.0.1 Specification, Companion Screen and Media Synchronization Sections*. Tech. rep. Online: http://www.etsi.org/deliver/etsi_ts/102700_102799/102796/01.04.01_60/ts_102796v010401p.pdf. Hybrid broadcast broadband TV (HbbTV), 2016 (cit. on pp. 12, 204).

[43]I. Fette and A. Melnikov. „The WebSocket Protocol“. In: (Dec. 2011). Online: https://tools.ietf.org/html/rfc6455 (cit. on pp. 13, 16, 88).

[44]*Bluetooth Low Energy*. Electronic Document. Online: https://www.bluetooth.com (cit. on pp. 13, 87, 104).

[45]*iBeacon*. Electronic Document. Online: https://developer.apple.com/ibeacon/ (cit. on p. 13).

[46]*The Physical Web*. Electronic Document. Online: https://google.github.io/physical-web/ (cit. on p. 13).

[47]Clement Vasseur. „Unofficial AirPlay Protocol Specification“. In: (Mar. 2012). Online: http://nto.github.io/AirPlay.html (cit. on p. 14).

[48]*MHL - Expand Your World*. Electronic Document. Online: http://www.mhltech.org/index.aspx (cit. on p. 15).

[49]R. Fielding, UC Irvine, J. Gettys, et al. „Hypertext Transfer Protocol – HTTP/1.1“. In: (Jan. 1997). Online: https://tools.ietf.org/html/rfc2068 (cit. on pp. 16, 88).

[50]J. Iyengar and M. Iyengar. „QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2“. In: (May 2018). Online: https://quicwg.github.io/base-drafts/draft-ietf-quic-transport.html (cit. on p. 16).

[51]„XMLHttpRequest API“. In: (May 2018). Online: https://xhr.spec.whatwg.org/ (cit. on pp. 16, 142).

[52]„Fetch API“. In: (May 2018). Online: https://fetch.spec.whatwg.org/ (cit. on pp. 16, 142).

[53]„HTML - WebSocket API“. In: (May 2018). Online: https://html.spec.whatwg.org/multipage/web-sockets.html (cit. on p. 16).

[54]H. Alvestrand. „Overview: Real Time Protocols for Browser-based Applications“. In: (Nov. 2017). Online: https://www.ietf.org/id/draft-ietf-rtcweb-overview-19.txt (cit. on pp. 16, 32, 34, 88, 109).

[55]Adam Bergkvist, Daniel Burnett, Cullen Jennings, et al. „WebRTC 1.0: Real-time Communication Between Browsers“. In: (Nov. 2017). Online: https://www.w3.org/TR/webrtc/ (cit. on p. 16).

[56]*Wi-Fi Direct*. Electronic Document. Online: https://www.wi-fi.org/discover-wi-fi/wi-fi-direct (cit. on p. 17).

[57]„H.264 : Advanced video coding for generic audiovisual services“. In: (Apr. 2017). Online: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-H.264-201704-I!!PDF-E (cit. on p. 18).

[58]„H.265 : High efficiency video coding“. In: (Feb. 2018). Online: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-H.265-201802-I!!PDF-E (cit. on pp. 18, 33).

[59]*VP9 Video Codec*. Electronic Document. Online: https://www.webmproject.org/vp9/ (cit. on p. 18).

[60]*A Large-Scale Comparison of x264, x265, and libvpx - a Sneak Peek*. Electronic Document. Online: https://medium.com/netflix-techblog/a-large-scale-comparison-of-x264-x265-and-libvpx-a-sneak-peek-2e81e88f8b0f (cit. on p. 18).

[61]Peter de Rivaz and Jack Haughton. „AV1 Bitstream and Decoding Process Specification“. In: (June 2018). Online: https://aomediacodec.github.io/av1-spec/av1-spec.pdf (cit. on p. 18).

[62]*ISO/IEC 14496-12:2015 Information technology - Coding of audio-visual objects - Part 12: ISO base media file format*. Standard Publication. Online: https://www.iso.org/standard/68960.html (cit. on pp. 19, 33).

[63]„Media Source Extensions MSE“. In: (Nov. 2016). Online: https://www.w3.org/TR/media-source/ (cit. on pp. 19, 23, 32).

[64]*ISO/IEC 13818-1:2018 Information technology - Generic coding of moving pictures and associated audio information - Part 1: Systems*. Standard Publication. Online: https://www.iso.org/standard/74427.html (cit. on p. 19).

[65]*ISO/IEC 23000-19:2018 Information technology - Multimedia application format (MPEG-A) - Part 19: Common media application format (CMAF) for segmented media*. Standard Publication. Online: https://www.iso.org/standard/71975.html (cit. on p. 19).

[66]*ISO/IEC FDIS 23090-2 Information technology - Coded representation of immersive media - Part 2: Omnidirectional media format*. Standard Publication. Online: https://www.iso.org/standard/73310.html (cit. on p. 20).

[67]*ISO/IEC FDIS 23009-1Information technology - Dynamic adaptive streaming over HTTP (DASH) - Part 1: Media presentation description and segment formats*. Standard Publication. Online: https://www.iso.org/standard/75485.html (cit. on pp. 20, 119, 146).

[68]„HTTP Live Streaming“. In: (Dec. 2011). Online: https://tools.ietf.org/html/rfc8216 (cit. on pp. 21, 119).

[69]„Nonlinear Projections“. In: *Transformations and Projections in Computer Graphics*. London: Springer London, 2006, pp. 145–220 (cit. on p. 21).

[70]Alain Galvan, Francisco Ortega, and Naphtali Rishe. „Procedural celestial rendering for 3D navigation“. In: *2017 IEEE Symposium on 3D User Interfaces (3DUI)*. Mar. 2017, pp. 211–212 (cit. on p. 21).

[71]Evgeny Kuzyakov and David Pio. *Under the hood: Building 360 video*. Blog. Online: https://code.fb.com/video-engineering/under-the-hood-building-360-video/ (cit. on p. 21).

[72]Evgeny Kuzyakov and David Pio. *Next-generation video encoding techniques for 360 video and VR*. Blog. Online: https://code.fb.com/virtual-reality/next-generation-video-encoding-techniques-for-360-video-and-vr/ (cit. on p. 22).

[73]Brandon Jones and Nell Waliczek. „WebXR Device API“. In: (Aug. 2018). Online: https://immersive-web.github.io/webxr/ (cit. on pp. 22, 34).

[74]Tatsuya Igarashi and Naoyuki Sato. „Expanding the Horizontal of Web“. In: *Third W3C Web and TV Workshop*. Online: https://www.w3.org/2011/09/webtv/papers/SONY_Position_Paper_3rdWebTVWorkshp_R0_1.pdf. Hollywood, California, USA, Sept. 2011 (cit. on pp. 24, 36).

[75]Clarke Stevens. „A Multi-protocol Home Networking Implementation for HTML5“. In: *Third W3C Web and TV Workshop*. Online: https://www.w3.org/2011/09/webtv/papers/W3C_HNTF_Position_Paper_Sept_2011.pdf. Hollywood, California, USA, Sept. 2011 (cit. on pp. 24, 36).

[76]W3C. *Network Service Discovery*. Technical report. Online: https://www.w3.org/TR/discovery-api/. W3C, Jan. 2017 (cit. on pp. 24, 36).

[77] Akitsugu Baba, Kinji Matsumura, Sigeaki Mitsuya, et al. „Advanced Hybrid Broadcast and Broadband System for Enhanced Broadcasting Services“. In: *NAB Broadcast Engineering Conference PROCEEDINGS*. Las Vegas, USA, Apr. 2011, pp. 343 –350 (cit. on p. 24).

[78] Maiko Imoto, Yasuhiko Miyazaki, Tetsuro Tokunaga, Kiyoshi Tanaka, and Shinji Miyahara. „A Framework for Supporting the Development of Multi-Screen Web Applications“. In: *Proceedings of International Conference on Information Integration and Web-based Applications and Services*. IIWAS '13. Vienna, Austria: ACM, 2013, 629:629–629:633 (cit. on pp. 24, 25, 36, 37).

[79] Hyojin Song, Soonbo Han, and Dong-Young Lee. „PARS - Multiscreen Web App Platform“. In: *Fourth W3C Web and TV Workshop*. Online: `https://www.w3.org/2013/10/tv-workshop/papers/webtv4_submission_9.pdf`. Munich, Germany, Mar. 2013 (cit. on pp. 25, 37).

[80] Jaejeung Kim, Sangtae Kim, and Howon Lee. „Partial Service/Application Migration and Device Adaptive User Interface across Multiple Screens“. In: *Third W3C Web and TV Workshop*. Online: `https://www.w3.org/2011/09/webtv/papers/W3C_3rd_WebTV_position_paper_KAIST_Final_submit.pdf`. Hollywood, California, USA, Sept. 2011 (cit. on pp. 25, 37).

[81] Jan Thomsen, el Troncy Rapha, and Nixon Lyndon. „Linking Web Content Seamlessly with Broadcast Television: Issues and Lessons Learned“. In: *Fourth W3C Web and TV Workshop*. Online: `https://www.w3.org/2013/10/tv-workshop/papers/webtv4_submission_15.pdf`. Munich, Germany, Mar. 2014 (cit. on p. 25).

[82] Raphaël Troncyl, Erik Mannens, Silvia Pfeiffer, and Davy Van Deursen. „Media Fragments URI 1.0“. In: (Sept. 2012). Online: `https://www.w3.org/TR/media-frags/` (cit. on p. 26).

[83] Njal Borch, Bin Cheng, Dave Raggett, and Mikel Zorrilla. „An architecture for second screen experiences based upon distributed social networks of people, devices and programs“. In: *Fourth W3C Web and TV Workshop*. Online: `https://www.w3.org/2013/10/tv-workshop/papers/webtv4_submission_6.pdf`. Munich, Germany, Mar. 2014 (cit. on pp. 26, 37).

[84] Geun-Hyung Kim and Sunghwan Kim. „Inter-Device Media Synchronization in Multi-Screen Environment“. In: *Fourth W3C Web and TV Workshop*. Online: `https://www.w3.org/2013/10/tv-workshop/papers/webtv4_submission_26.pdf`. Munich, Germany, Mar. 2014 (cit. on p. 26).

[85] Victor Klos. „Three Challenges for Web&TV“. In: *Fourth W3C Web and TV Workshop*. Online: `https://www.w3.org/2013/10/tv-workshop/papers/webtv4_submission_12.pdf`. Munich, Germany, Mar. 2014 (cit. on pp. 26, 37).

[86] C. Howson, E. Gautier, P. Gilberton, A. Laurent, and Y. Legallais. „Second screen TV synchronization“. In: *2011 IEEE International Conference on Consumer Electronics -Berlin (ICCE-Berlin)*. Sept. 2011, pp. 361–365 (cit. on p. 26).

[87] Paul Tolstoi and Andreas Dippon. „Towering Defense: An Augmented Reality Multi-Device Game“. In: *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*. CHI EA '15. Seoul, Republic of Korea: ACM, 2015, pp. 89–92 (cit. on pp. 27, 37).

[88] Mira Sarkis, Cyril Concolato, and Jean-Claude Dufourd. „A multi-screen refactoring system for video-centric web applications“. In: *Multimedia Tools and Applications* (Jan. 2017) (cit. on pp. 27, 37).

[89] Bongjin Oh and Park Jongyoul. „A remote user interface framework for collaborative services using globally internetworked smart appliances“. In: *2015 17th International Conference on Advanced Communication Technology (ICACT)*. July 2015, pp. 581–586 (cit. on pp. 27, 37).

[90] Yichao Jin, Tian Xie, Yonggang Wen, and Haiyong Xie. „Multi-screen Cloud Social TV: Transforming TV Experience into 21st Century“. In: *Proceedings of the 21st ACM International Conference on Multimedia*. MM '13. Barcelona, Spain: ACM, 2013, pp. 435–436 (cit. on pp. 27, 28, 37).

[91] Michael Krug, Fabian Wiedemann, and Martin Gaedke. „SmartComposition: A Component-Based Approach for Creating Multi-screen Mashups“. In: *Web Engineering: 14th International Conference, ICWE 2014, Toulouse, France, July 1-4, 2014. Proceedings*. Ed. by Sven Casteleyn, Gustavo Rossi, and Marco Winckler. Cham: Springer International Publishing, 2014, pp. 236–253 (cit. on pp. 28, 37).

[92] European Commission : CORDIS : Programmes : Specific Programme "Cooperation": Information and communication technologies. *Open Mashup Enterprise service platform for LinkEd data in The TElco domain*. 2013 (cit. on p. 28).

[93] Francisco Martinez-Pabon, Jaime Caicedo-Guerrero, Jhon Jairo Ibarra-Samboni, Gustavo Ramirez-Gonzalez, and Davinia Hernández-Leo. „Smart TV-Smartphone Multi-screen Interactive Middleware for Public Displays“. In: *The Scientific World Journal* 2015 (Apr. 2015), p. 534949 (cit. on p. 28).

[94] Changwoo Yoon, Taiwon Um, and Hyunwoo Lee. „Classification of N-Screen Services and its standardization“. In: *2012 14th International Conference on Advanced Communication Technology (ICACT)*. Feb. 2012, pp. 597–602 (cit. on p. 28).

[95] Xinfeng Xie, Zhongqing Yu, and Kaixi Wang. „The design and implementation of the multi-screen interaction service architecture for the Real-Time streaming media“. In: *2013 Ninth International Conference on Natural Computation (ICNC)*. July 2013, pp. 1600–1604 (cit. on pp. 28, 37).

[96] Dong-Hoon Lee, Jung-Hyun Kim, Ho-Youn Kim, and Dong-Young Park. „Remote Application Control Technology and Implementation of HTML5-based Smart TV Platform“. In: *Proceedings of the 14th International Conference on Advances in Mobile Computing and Multi Media*. MoMM '16. Singapore, Singapore: ACM, 2016, pp. 208–211 (cit. on p. 29).

[97] Jorge Abreu, Pedro Almeida, and Telmo Silva. „Enriching Second-Screen Experiences with Automatic Content Recognition“. In: *VI International Conference on Interactive Digital TV IV Iberoamerican Conference on Applications and Usability of Interactive TV*. 2015, pp. 41–50 (cit. on p. 29).

[98] Ui Nyoung Yoon, Seung Hyun Ko, Kyeong-Jin Oh, and Geun-Sik Jo. „Thumbnail-based interaction method for interactive video in multi-screen environment“. In: *2016 IEEE International Conference on Consumer Electronics (ICCE)*. Jan. 2016, pp. 3–4 (cit. on pp. 29, 37).

[99] M. Punt. „Rebooting the TV-centric gaming concept for modern multiscreen Over-The-Top service“. In: *2016 Zooming Innovation in Consumer Electronics International Conference (ZINC)*. June 2016, pp. 50–54 (cit. on pp. 30, 37).

[100] Pedro Centieiro, Teresa Romão, and A. Eduardo Dias. „Enhancing Remote Spectators' Experience During Live Sports Broadcasts with Second Screen Applications“. In: *More Playful User Interfaces: Interfaces that Invite Social and Physical Interaction*. Ed. by Anton Nijholt. Singapore: Springer Singapore, 2015, pp. 231–261 (cit. on pp. 30, 37).

[101] David Geerts, Rinze Leenheer, Dirk De Grooff, Joost Negenman, and Susanne Heijstraten. „In Front of and Behind the Second Screen: Viewer and Producer Perspectives on a Companion App“. In: *Proceedings of the ACM International Conference on Interactive Experiences for TV and Online Video*. TVX '14. Newcastle Upon Tyne, United Kingdom: ACM, 2014, pp. 95–102 (cit. on pp. 30, 38).

[102] Vinod Keshav Seetharamu, Joy Bose, Sowmya Sunkara, and Nitesh Tigga. „TV remote control via wearable smart watch device“. In: *2014 Annual IEEE India Conference (INDICON)*. Dec. 2014, pp. 1–6 (cit. on p. 31).

[103] Thomas Stockhammer. „Dynamic Adaptive Streaming over HTTP –: Standards and Design Principles“. In: *Proceedings of the Second Annual ACM Conference on Multimedia Systems*. MMSys '11. San Jose, CA, USA: ACM, 2011, pp. 133–144 (cit. on p. 31).

[104] Omar A. Niamut, Emmanuel Thomas, Lucia D'Acunto, et al. „MPEG DASH SRD: Spatial Relationship Description“. In: *Proceedings of the 7th International Conference on Multimedia Systems*. MMSys '16. Klagenfurt, Austria: ACM, 2016, 5:1–5:8 (cit. on pp. 32, 38, 120).

[105] Volker Jung, Stefan Pham, and Stefan Kaiser. „A web-based media synchronization framework for MPEG-DASH“. In: *2014 IEEE International Conference on Multimedia and Expo Workshops (ICMEW)*. July 2014, pp. 1–2 (cit. on pp. 32, 38).

[106] Mohammad Hosseini and Viswanathan Swaminathan. „Adaptive 360 VR Video Streaming Based on MPEG-DASH SRD“. In: *2016 IEEE International Symposium on Multimedia (ISM)*. Dec. 2016, pp. 407–408 (cit. on pp. 32, 33, 38).

[107] Cyril Concolato, Jean Le Feuvre, Franck Denoual, et al. „Adaptive Streaming of HEVC Tiled Videos using MPEG-DASH“. In: *IEEE Transactions on Circuits and Systems for Video Technology* PP.99 (2017), pp. 1–1 (cit. on pp. 33, 120).

[108] Ray Van Brandenburg, Omar Niamut, Martin Prins, and Hans Stokking. „Spatial segmentation for immersive media delivery“. In: *2011 15th International Conference on Intelligence in Next Generation Networks*. Oct. 2011, pp. 151–156 (cit. on pp. 33, 38).

[109] Omar A. Niamut, Axel Kochale, Javier Ruiz Hidalgo, et al. „Towards a Format-agnostic Approach for Production, Delivery and Rendering of Immersive Media“. In: *Proceedings of the 4th ACM Multimedia Systems Conference*. MMSys '13. Oslo, Norway: ACM, 2013, pp. 249–260 (cit. on pp. 33, 38).

[110] Aditya Mavlankar, Jeonghun Noh, Pierpaolo Baccichet, and Bernd Girod. „Peer-to-peer multicast live video streaming with interactive virtual pan/tilt/zoom functionality“. In: *2008 15th IEEE International Conference on Image Processing*. Oct. 2008, pp. 2296–2299 (cit. on p. 33).

[111]Yonggang Wen, Xiaoqing Zhu, Joel J. P. C. Rodrigues, and Chang Wen Chen. „Cloud Mobile Media: Reflections and Outlook“. In: *IEEE Transactions on Multimedia* 16.4 (June 2014), pp. 885–902 (cit. on p. 34).

[112]Alireza Zare, Alireza Aminlou, Miska M. Hannuksela, and Moncef Gabbouj. „HEVC-compliant Tile-based Streaming of Panoramic Video for Virtual Reality Applications“. In: *Proceedings of the 2016 ACM on Multimedia Conference*. MM '16. Amsterdam, The Netherlands: ACM, 2016, pp. 601–605 (cit. on pp. 34, 39).

[113]Yichao Jin, Yonggang Wen, Han Hu, and Marie-Jose Montpetit. „Reducing Operational Costs in Cloud Social TV: An Opportunity for Cloud Cloning“. In: *IEEE Transactions on Multimedia* 16.6 (Oct. 2014), pp. 1739–1751 (cit. on p. 34).

[114]Niklas Carlsson, Derek Eager, Krishnamoorthi Vengatanathan, and Tatiana Polishchuk. „Optimized Adaptive Streaming of Multi-video Stream Bundles“. In: *IEEE Transactions on Multimedia* 19.7 (July 2017), pp. 1637–1653 (cit. on p. 34).

[115]Simon Gunkel, Martin Prins, Hans Stokking, and Omar Niamut. „WebVR meets WebRTC: Towards 360-degree social VR experiences“. In: *2017 IEEE Virtual Reality (VR)*. Mar. 2017, pp. 457–458 (cit. on p. 34).

[116]RG Belleman, B Stolk, R de Vries, et al. „Immersive Virtual Reality on commodity hardware“. In: ASCI. 2001 (cit. on p. 35).

[117]Feng Qian, Lusheng Ji, Bo Han, and Vijay Gopalakrishnan. „Optimizing 360 Video Delivery over Cellular Networks“. In: *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*. ATC '16. New York City, New York: ACM, 2016, pp. 1–6 (cit. on pp. 35, 39).

[118]Luís A. R. Neng and Teresa Chambel. „Get Around 360&Deg; Hypervideo“. In: *Proceedings of the 14th International Academic MindTrek Conference: Envisioning Future Media Environments*. MindTrek '10. Tampere, Finland: ACM, 2010, pp. 119–122 (cit. on p. 35).

[119]Derek Pang, Sherif Halawa, Ngai-Man Cheung, and Bernd Girod. „Mobile Interactive Region-of-interest Video Streaming with Crowd-driven Prefetching“. In: *Proceedings of the 2011 International ACM Workshop on Interactive Multimedia on Mobile and Portable Devices*. IMMPD '11. Scottsdale, Arizona, USA: ACM, 2011, pp. 7–12 (cit. on p. 35).

[120]Daisuke Ochi, Yutaka Kunita, Kensaku Fujii, et al. „HMD Viewing Spherical Video Streaming System“. In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. MM '14. Orlando, Florida, USA: ACM, 2014, pp. 763–764 (cit. on pp. 36, 39).

[121]Rovio Entertainment Corporation. *Angry Birds*. Electronic Document. Online: `http://www.rovio.com/games/angry-birds` (cit. on p. 43).

[122]*DLNA*. Electronic Document. Online: `https://www.dlna.org` (cit. on p. 51).

[123]Arthur Gill et al. „Introduction to the theory of finite-state machines“. In: (1962) (cit. on p. 60).

[124]Rajeev Alur and David L Dill. „A theory of timed automata“. In: *Theoretical computer science* 126.2 (1994), pp. 183–235 (cit. on p. 60).

[125]Heiko Pfeffer, Louay Bassbouss, and Stephan Steglich. „Structured Service Composition Execution for Mobile Web Applications“. In: *2008 12th IEEE International Workshop on Future Trends of Distributed Computing Systems*. Kunming, China, Oct. 2008, pp. 112–118 (cit. on pp. 60, 203, 204).

[126]N. E. Baughman and B. N. Levine. „Cheat-proof playout for centralized and distributed online games“. In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*. Vol. 1. 2001, 104–113 vol.1 (cit. on pp. 77, 113).

[127]Nir Shavit and Dan Touitou. „Software Transactional Memory“. In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '95. Ottowa, Ontario, Canada: ACM, 1995, pp. 204–213 (cit. on p. 78).

[128]Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. „Software Transactional Memory for Dynamic-sized Data Structures“. In: *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*. PODC '03. Boston, Massachusetts: ACM, 2003, pp. 92–101 (cit. on p. 78).

[129]L. Gautier, C. Diot, and J. Kurose. „End-to-end transmission control mechanisms for multiparty interactive applications on the Internet“. In: *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*. Vol. 3. 1999, 1470–1479 vol.3 (cit. on pp. 78, 113).

[130]D. L. Mills. „Internet time synchronization: the network time protocol“. In: *IEEE Transactions on Communications* 39.10 (1991), pp. 1482–1493 (cit. on p. 78).

[131]D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. „Time Warp Operating System“. In: *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*. SOSP '87. Austin, Texas, USA: ACM, 1987, pp. 77–93 (cit. on p. 78).

[132]Eric Cronin, Burton Filstrup, Anthony R. Kurc, and Sugih Jamin. „An Efficient Synchronization Mechanism for Mirrored Game Architectures“. In: *Proceedings of the 1st Workshop on Network and System Support for Games*. NetGames '02. Braunschweig, Germany: ACM, 2002, pp. 67–73 (cit. on pp. 78, 113).

[133]*Cloud Browser Architecture*. Technical Report. Online: `https://www.w3.org/TR/cloud-browser-arch/`. The World Wide Web Consortium (W3C), 2017 (cit. on pp. 82, 162).

[134]„Stadia | Build a new generation of games“. In: (2019). Online: `https://stadia.dev` (cit. on pp. 82, 158, 167).

[135]„The JSON Data Interchange Syntax“. In: (Dec. 2017). Online: `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf` (cit. on p. 89).

[136]„Custom elements“. In: (Sept. 2018). Online: `https://html.spec.whatwg.org/multipage/custom-elements.html#custom-elements` (cit. on p. 93).

[137]W3C. *Web Platform Working Group*. Tech. rep. Online: `https://www.w3.org/WebPlatform/WG/`. The World Wide Web Consortium (W3C), 2018 (cit. on p. 93).

[138]„Shadow Tree“. In: (Aug. 2018). Online: `https://dom.spec.whatwg.org/#shadow-trees` (cit. on p. 93).

[139]„The template element“. In: (Sept. 2018). Online: https://html.spec.whatwg.org/multipage/scripting.html#the-template-element (cit. on p. 93).

[140]*peer-ssdp: Node.js Implementation of the Simple Service Discovery Protocol SSDP*. Open Source Implementation. Online: https://github.com/fraunhoferfokus/peer-ssdp/. Fraunhofer FOKUS, 2017 (cit. on pp. 103, 206).

[141]*cordova-plugin-hbbtv: Cordova Plugin Implemenation of the HbbTV Companion Screen Specification*. Open Source Implementation. Online: https://github.com/fraunhoferfokus/cordova-plugin-hbbtv. Fraunhofer FOKUS, 2017 (cit. on pp. 103, 207).

[142]„UserNotifications“. In: (2018). Online: https://developer.apple.com/documentation/usernotifications (cit. on p. 105).

[143]Alberto Montresor. „Gossip and epidemic protocols“. In: *Wiley Encyclopedia of Electrical and Electronics Engineering* (1999), pp. 1–15 (cit. on p. 111).

[144]„WebView | Android Developers“. In: (2019). Online: https://developer.android.com/reference/android/webkit/WebView (cit. on p. 113).

[145]CEF Open Source Community. *Chromium Embedded Framework*. Electronic Document. Online: https://bitbucket.org/chromiumembedded/cef (cit. on pp. 113, 115).

[146]T. Berners-Lee, R. Fielding, and L. Masinter. „https://tools.ietf.org/html/rfc3986“. In: (Jan. 2005). Online: https://tools.ietf.org/html/rfc3986 (cit. on p. 113).

[147]„Uniform Resource Identifier (URI) Schemes“. In: (2019). Online: https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml (cit. on p. 113).

[148]Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. „Overview of the High Efficiency Video Coding (HEVC) Standard“. In: *IEEE Trans. Cir. and Sys. for Video Technol.* 22.12 (Dec. 2012), pp. 1649–1668 (cit. on p. 120).

[149]Flaviu Cristian. „Probabilistic clock synchronization“. In: *Distributed Computing* 3.3 (1989), pp. 146–158 (cit. on p. 125).

[150]David L. Mills. „A Brief History of NTP Time: Memoirs of an Internet Timekeeper“. In: *SIGCOMM Comput. Commun. Rev.* 33.2 (Apr. 2003), pp. 9–21 (cit. on p. 125).

[151]Louay Bassbouss, Stephan Steglich, and Christian Fuhrhop. „Smart TV 360“. In: *Broadcast Engineering and Information Technology Conference, Virtual and Augmented Reality/Immersive Content*. Las Vegas, USA, 2017 (cit. on pp. 127, 202).

[152]*A Tour of the West (1955)*. Electronic Document. Online: http://www.imdb.com/title/tt0048742/ (cit. on p. 127).

[153]*FUTURE MATTERS - Circle-Vision 360 - Imagineering Disney* -. Electronic Document. Online: http://www.imagineeringdisney.com/blog/2016/10/6/future-matters-circle-vision-360.html (cit. on p. 127).

[154]hiow Keng Tan, Rajitha Weerakkody, Marta Mrak, et al. „Video Quality Evaluation Methodology and Verification Testing of HEVC Compression Performance“. In: *IEEE Transactions on Circuits and Systems for Video Technology* 26.1 (2016), pp. 76–90 (cit. on p. 129).

[155]*Recommended upload encoding settings - YouTube Help*. Electronic Document. Online: https://support.google.com/youtube/answer/172217 (cit. on pp. 130, 164).

[156]M. P. Sharabayko and N. G. Markov. „Contemporary video compression standards: H.265/HEVC, VP9, VP10, Daala“. In: *2016 International Siberian Conference on Control and Communications (SIBCON)*. 2016, pp. 1–4 (cit. on p. 130).

[157]Miroslav Uhrina, Juraj Bienik, and Martin Vaculik. „Coding efficiency of HEVC/H.265 and VP9 compression standards for high resolutions“. In: *2016 26th International Conference Radioelektronika (RADIOELEKTRONIKA)*. 2016, pp. 419–423 (cit. on p. 130).

[158]Bappaditya Ray, Joel Jung, and Mohamed-Chaker Larabi. „A Low-Complexity Video Encoder for Equirectangular Projected 360 Video Content“. In: *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2018, pp. 1723–1727 (cit. on p. 131).

[159]Blender Institute. *Caminandes VR Demo*. Electronic Document. Online: `https://cloud.blender.org/p/caminandes-3/blog/caminandes-llamigos-vr-demo` (cit. on p. 133).

[160]Blender Institute. *Caminandes VR Demo - YouTube*. Online: `https://www.youtube.com/watch?v=uvy--ElpfF8`. 2016 (cit. on p. 133).

[161]„Biathlon Worldcup live in 360°“. In: (2019). Online: `https://www.fokus.fraunhofer.de/en/fame/biathlon360` (cit. on p. 139).

[162]„Smart TV: Die ZDFmediathek auf Ihrem TV-Gerät - ZDFmediathek“. In: (2019). Online: `https://www.zdf.de/service-und-hilfe/zdf-mediathek/smarttv-100.html` (cit. on p. 139).

[163]Didier Le Gall. „MPEG: A Video Compression Standard for Multimedia Applications“. In: *Commun. ACM* 34.4 (Apr. 1991), pp. 46–58 (cit. on p. 139).

[164]Martin Lasak, Louay Bassbouss, and Stephan Steglich. *[DE] Verarbeitungsverfahren und Verarbeitungssystem für Videodaten*. Patent. Patent Number: DE 102017125544B3, Published June 28, 2018, Online: `https://depatisnet.dpma.de/DepatisNet/depatisnet?action=bibdat&docid=DE102017125544B3`. June 2018 (cit. on pp. 145, 204).

[165]Martin Lasak, Louay Bassbouss, and Stephan Steglich. *Processing Method and Processing System for Video Data*. Patent. Patent Number: WO2018210485, Published November 22, 2018, Online: `https://patentscope.wipo.int/search/en/detail.jsf?docId=WO2018210485`. Nov. 2018 (cit. on pp. 145, 204).

[166]Blender Foundation. *Big Buck Bunny*. Electronic Document. Online: `https://peach.blender.org/` (cit. on p. 151).

[167]Stephen Perrott. *MPEG DASH Test Streams*. Electronic Document. Online: `http://www.bbc.co.uk/rd/blog/2013-09-mpeg-dash-test-streams`. 2013 (cit. on p. 152).

[168]„How to use Activity Monitor on your Mac“. In: (2019). Online: `https://support.apple.com/en-us/HT201464#energy` (cit. on p. 156).

[169]„Biathlon Worldcup live in 360°“. In: (2019). Online: `https://www.fokus.fraunhofer.de/en/fame/biathlon360` (cit. on p. 166).

[170]„Stadia Founder's Edition“. In: (2019). Online: `https://store.google.com/product/stadia_founders_edition` (cit. on p. 167).

[171] Avraham Leff and James Rayfield. „Web-application development using the Model/View/Controller design pattern“. In: *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*. 2001, pp. 118–127 (cit. on p. 169).

[172] Louay Bassbouss, Stephan Steglich, and Igor Fritzsch. „Interactive 360° Video and Storytelling Tool“. In: *2019 IEEE 23rd International Symposium On Consumer Technologies (IEEE ISCT2019)*. Ancona, Italy, 2019 (cit. on p. 201).

[173] Louay Bassbouss, Stefan Pham, Stephan Steglich, and Martin Lasak. „Content Preparation and Cross-Device Delivery of 360° Video with 4K Field of View using DASH“. In: *2017 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*. Hong Kong, 2017 (cit. on p. 201).

[174] Paul Murdock, Louay Bassbouss, Martin Bauer, et al. *Semantic interoperability for the Web of Things*. White Paper. Online: `http://dx.doi.org/10.13140/RG.2.2.25758.13122`. IEEE Standards Association, AIOTI, oneM2M and W3C Joint Collaboration, Aug. 2016 (cit. on p. 202).

[175] Louay Bassbouss and Stephan Steglich. „Position Paper: High quality 360° Video Rendering and Streaming on the Web“. In: *W3C Workshop on Web and Virtual Reality*. San Jose, CA, USA, 2016 (cit. on p. 202).

[176] Louay Bassbouss. *Einführung in das Physical Web*. Electronic Document. Online: `https://heise.de/-2919078`. Heise Developer, 2015 (cit. on p. 202).

[177] Louay Bassbouss, Görkem Güçlü, and Stephan Steglich. „Towards a remote launch mechanism of TV companion applications using iBeacon“. In: *2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE)*. Tokyo, Japan, 2014, pp. 538–539 (cit. on p. 202).

[178] Christopher Krauss, Louay Bassbouss, Stefan Pham, et al. „Position Paper: Challenges for enabling targeted multi-screen advertisement for interactive TV services“. In: *W3C Web and TV Workshop*. Munich, Germany, 2014 (cit. on p. 202).

[179] Jean-Claude Dufourd, Louay Bassbouss, Max Tritschler, Radhouane Bouazizi, and Stephan Steglich. „An Open Platform for Multiscreen Services“. In: *EuroITV 2013: 11th European Interactive TV Conference*. Como, Italy, 2013 (cit. on p. 203).

[180] Evanela Lapi, Nikolay Tcholtchev, Louay Bassbouss, Florian Marienfeld, and Ina Schieferdecker. „Identification and Utilization of Components for a Linked Open Data Platform“. In: *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*. Izmir, Turkey, 2012, pp. 112–115 (cit. on p. 203).

[181] Robert Kleinfeld, Louay Bassbouss, Iosif Alvertis, and George Gionis. „Empowering Civic Participation in the Policy Making Process through Social Media“. In: *International AAAI Conference on Web and Social Media*. Dublin, Ireland, 2012 (cit. on p. 203).

[182] George Gionis, Louay Bassbouss, Heïko Desruelle, et al. „'Do we know each other or is it just our devices?': a federated context model for describing social activity across devices“. eng. In: *Federated Social Web Europe 2011, Proceedings*. Berlin, Germany: W3C ; PrimeLife, 2011, p. 6 (cit. on p. 203).

[183] Heiko Pfeffer, Louay Bassbouss, David Linner, et al. „Mixing Workflows and Components to Support Evolving Services“. In: *International Journal of Adaptive, Resilient and Autonomic Systems* 1.4 (2010), pp. 60–84 (cit. on p. 203).

[184] Iacopo Carreras, Louay Bassbouss, David Linner, et al. „BIONETS: Self Evolving Services in Opportunistic Networking Environments“. In: *Bioinspired Models of Network, Information, and Computing Systems: 4th International Conference, BIONETICS 2009, Avignon, France, December 9-11, 2009*. Ed. by Eitan Altman, Iacopo Carrera, Rachid El-Azouzi, Emma Hart, and Yezekael Hayel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 88–94 (cit. on p. 203).

[185] W3C. *Media and Entertainment Interest Group*. Tech. rep. Online: `https://www.w3.org/2011/webtv/`. The World Wide Web Consortium (W3C), 2017 (cit. on p. 204).

[186] W3C. *Web of Things*. Tech. rep. Online: `https://www.w3.org/WoT/`. The World Wide Web Consortium (W3C), 2017 (cit. on p. 204).

[187] Igor Fritzsch. „360° Storytelling:Mixed Media, Analytics and Interaction Design“. MA thesis. Technical University of Berlin, June 2019 (cit. on p. 205).

[188] Thomas Fett. „Design and Implementation of a Sound Engine for 360° Videos in Web Browsers and Smart TVs“. MA thesis. Technical University of Berlin, Dec. 2018 (cit. on p. 205).

[189] Christian Bach. „360° Video Streaming for Head-Mounted Displays“. MA thesis. Wildau Technical University of Applied Science, June 2018 (cit. on p. 205).

[190] Marius Wessel. „Assembler on the Web - Evaluation of the WebAssembly Technology“. MA thesis. Technical University of Berlin, June 2018 (cit. on p. 205).

[191] Lukas Rögner. „Cloud-Based Application Rendering for Low-Capability Devices“. MA thesis. Technical University of Berlin, Oct. 2017 (cit. on p. 205).

[192] Christian Bromann. „Design and Implementation of a Development and Test Automation Platform for HbbTV“. MA thesis. Technical University of Berlin, Aug. 2017 (cit. on p. 205).

[193] Jonas Rook. „Konzipierung und Entwicklung eines W3C konformen Web of Things Framework“. MA thesis. HTW Berlin, Mar. 2017 (cit. on p. 205).

[194] Akshay Akshay. „Analysis and Implementation of Unified Synchronization Framework for HbbTV2.0 Sync-API and W3C Web-Timing API“. MA thesis. Kiel University of Applied Sciences, Feb. 2016 (cit. on p. 205).

[195] Tommy Weidt. „Synchronization Framework for W3C Second Screen Presentation API“. MA thesis. Technical University of Berlin, June 2015 (cit. on p. 205).

[196] Yi Fan. „Platform for sharing and synchronization of web content in multiscreen applications“. MA thesis. Technical University of Berlin, Jan. 2015 (cit. on p. 205).

[197] Kostiantyn Kahanskyi. „Dynamic Media Objects“. MA thesis. Technical University of Berlin, Apr. 2014 (cit. on p. 205).

[198] Anne Haase. „Design and implementation of a migration framework for multiscreen applications“. MA thesis. Free University of Berlin, Jan. 2014 (cit. on p. 205).

[199] Lutz Welpelo. „Plattform zur Verfolgung von Produkt- und Markenpiraterie auf Online-Marktplätzen“. MA thesis. Technical University of Berlin, July 2013 (cit. on p. 206).

[200] Alexander Futasz. „Web Scraping Cloud Platform With Integrated Visual Editor and Runtime Environment“. MA thesis. Technical University of Berlin, Mar. 2013 (cit. on p. 206).

[201] Michal Radziwonowicz. „Development and Cross-domain Runtime Environment for Distributed Mashups“. MA thesis. Technical University of Berlin, Jan. 2013 (cit. on p. 206).

[202] Ahmad Abbas. „Cloud Platform for Web Connected Sensors and Actuators“. MA thesis. Beuth University of Applied Sciences Berlin, Dec. 2012 (cit. on p. 206).

[203] Niklas Schmücker. „Enhancing Web-Based Citizen Reporting Platforms for the Public Sector through Social Media“. MA thesis. Technical University of Berlin, Feb. 2012 (cit. on p. 206).

[204] Hui Deng. „Click-By-Click Mashup Platform for Open Statistical Data“. MA thesis. Technical University of Berlin, Apr. 2011 (cit. on p. 206).

[205] Alexander Kong. „Securing Semi-automatic Data flow Control in Government Mashups“. MA thesis. Technical University of Berlin, Dec. 2010 (cit. on p. 206).

[206] Jie Lu. „Towards an End-User Centric Mashup Creation Environment facilitated through Code Sharing“. MA thesis. Technical University of Berlin, June 2010 (cit. on p. 206).

[207] *peer-upnp: Node.js Implementation of the Universal Plug and Play Protocol UPnP*. Open Source Implementation. Online: `https://github.com/fraunhoferfokus/peer-upnp/`. Fraunhofer FOKUS, 2017 (cit. on p. 206).

[208] *peer-dial: Node.js Implementation of the Discovery and Launch Protocol DIAL*. Open Source Implementation. Online: `https://github.com/fraunhoferfokus/peer-dial/`. Fraunhofer FOKUS, 2017 (cit. on p. 207).

[209] *node-hbbtv: Node.js Implementation of the HbbTV Companion Screen Specification*. Open Source Implementation. Online: `https://github.com/fraunhoferfokus/node-hbbtv/`. Fraunhofer FOKUS, 2017 (cit. on p. 207).

[210] *cordova-plugin-presentation: Cordova Plugin Implemenation of the W3C Second Screen Presentation API for Airplay and Miracast*. Open Source Implementation. Online: `https://github.com/fraunhoferfokus/cordova-plugin-presentation`. Fraunhofer FOKUS, 2017 (cit. on p. 207).

[211] *Concept and Implementation of UPnP/SSDP Support in Physical Web*. Open Source Implementation. Online: `https://github.com/google/physical-web/blob/master/documentation/ssdp_support.md`. Google, 2017 (cit. on p. 207).

[212] Louay Bassbouss and Christopher Krauß. *Personalized Multi-Platform Development*. Guest Lecture. Beuth University of Applied Sciences Berlin, Feb. 2015 (cit. on p. 207).

[213] Louay Bassbouss. *Multiscreen Technologies, Standards and Best Practices*. Guest Lecture. Beuth University of Applied Sciences Berlin, May 2015 (cit. on p. 207).

[214] Louay Bassbouss. *Multiscreen Technologies and Standards*. Guest Lecture. Beuth University of Applied Sciences Berlin, Jan. 2017 (cit. on p. 207).

[215] Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Lecture WS 2015/2016*. University Course. Technical University Berlin, 2015 (cit. on p. 207).

[216] Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Project SS 2016*. University Course. Technical University Berlin, 2016 (cit. on p. 208).

[217] Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Lecture WS 2016/2017*. University Course. Technical University Berlin, 2016/17 (cit. on p. 208).

[218] Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Project WS 2016/2017*. University Course. Technical University Berlin, 2016/17 (cit. on p. 208).

[219] Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Project SS 2017*. University Course. Technical University Berlin, 2017 (cit. on p. 208).

[220] Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Lecture WS 2017/2018*. University Course. Technical University Berlin, 2017/18 (cit. on p. 208).

[221] Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Project WS 2017/2018*. University Course. Technical University Berlin, 2017/18 (cit. on p. 208).

[222] Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Project SS 2018*. University Course. Technical University Berlin, 2018 (cit. on p. 208).

List of Figures

List of Tables

Acronyms

AAC Atomic Application Component
ABR Adaptive Bitrate
ACR Automatic Content Recognition
API Application Programming Interface
APN Apple Push Notification
APNs Apple Push Notification service
AR Augmented Reality
AVC Advanced Video Coding
AWS Amazon Web Services
BLE Bluetooth Low Energy
CAC Composite Application Component
CDN Content Distribution Network
CEF Chrome Embedded Framework
CG Community Group
CMAF Common Media Application Format
CPU Central Processing Unit
CS Companion Screen
CSP Client Side Processing
CSS Cascading Style Sheets
CST Client Side Transformation
DASH Dynamic Adaptive Streaming over HTTP
DDR Double Data Rate
DIAL Discovery and Launch protocol
DLNA Digital Living Network Alliance
DNS Domain Name System
DNS-SD DNS Service Discovery
DOM Document Object Model
DRM Digital Rights Management
EME Encrypted Media Extensions
EPG Electronic Program Guide
EQR Equirectangular

FHD Full High Definition
FMC Fixed-Mobile Convergence
FOV Field Of View
FPS Frames Per Second
GOP Group Of Pictures
GPS Global Positioning System
GPU Graphics Processing Unit
HbbTV Hybrid broadcast broadband TV
HD High Definition
HDMI High-Definition Multimedia Interface
HEVC High-Efficiency Video Coding
HLS HTTP Live Streaming
HMD Head-Mounted Display
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
IEC International Electrotechnical Commission
IEEE Institute of Electrical and Electronics Engineers
IO Input Output
IP Internet Protocol
ISO International Organization for Standardization
ISOBMFF ISO Base Media File Format
ITU International Telecommunication Union
JPEG Joint Photographic Experts Group
JSON JavaScript Object Notation
MB Megabyte
mDNS multicast Domain Name System
MHL Mobile High-Definition Link
MPD Media Presentation Description
MPEG Moving Picture Experts Group
MSA Multiscreen Application
MSC Multiscreen Application Component
MSE Media Source Extension
MVC Model View Controller
NAB National Association of Broadcasters
NAT Network Address Translation
NFC Near Field Communication
NTP Network Time Protocol
OMAF Omnidirectional MediA Format
OMDL Open Mashup Description Language
OS Operating System
OTT Over The Top
PC Personal Computer

PNG Portable Network Graphics
PSNR Peak Signal-to-Noise Ratio
PTR Pointer Record
PTZ Pan Tilt Zoom
QR Quick Response
QUIC Quick UDP Internet Connections
RCP Remote Control Protocol
REQ Requirement
REST Representational State Transfer
ROI Region Of Interest
RPC Remote Procedure Call
RTC Real Time Communication
RTSP Real Time Streaming Protocol
RTT Round Trip Time
RUI Remote User Interface
SD Standard Definition
SDK Software Development Kit
SRD Spacial Relationship Description
SRN Segment Recombination Node
SRV Service Record
SSDP Simple Service Discovery Protocol
SSP Service Side Processing
SST Service Side Transformation
STB Set Top Box
STUN Session Traversal Utilities for NAT
TCP Transmission Control Protocol
TS Transport Stream
TTL Time To Live
TURN Traversal Using Relay NAT
TV Television
TXT Text Record
UA User Agent
UC Use Case
UDP User Datagram Protocol
UHD Ultra High Definition
UI User Interface
UML Unified Modeling Language
UPNP Universal Plug and Play
URI Uniform Resource Identifier
URL Uniform Resource Locator
UUID Universally Unique Identifier
VOD Video On Demand

VR Virtual Reality
W3C World Web Consortium
WAMP Web Application Message Protocol
WG Working Group
WIPO World Intellectual Property Organization
WS WebSockets
XHR XMLHttpRequest
XML eXtensible Markup Language
XMPP Extensible Messaging and Presence Protocol

Appendices

Author's Publications

A

This chapter summarizes all contributions of the author during this thesis. Section A.1 lists all accepted and published papers to national and international conferences, journals and events. Section A.2 lists an accepted patent about a new mechanism for a smooth transition between different perspectives in videos which is related to the 360° pre-rendering solution introduced in this work. Section A.3 lists all contributions to relevant standards and section A.4 lists all diploma, bachelor and master theses supervised by the author of this work. the author's open source contributions related to the topic of this thesis are listed in section A.5. Finally, author's contributions to university courses and guest lectures are listed in section A.6.

A.1 Accepted Papers and Published Articles

1. Louay Bassbouss, Stephan Steglich, and Igor Fritzsch. „Interactive 360° Video and Storytelling Tool". In: *2019 IEEE 23rd International Symposium On Consumer Technologies (IEEE ISCT2019)*. Ancona, Italy, 2019 [172]

2. Louay Bassbouss, Stefan Pham, and Stephan Steglich. „Streaming and Playback of 16K 360° Videos on the Web". In: *2018 IEEE Middle East and North Africa Communications Conference (MENACOMM) (IEEE MENACOMM'18)*. Jounieh, Lebanon, 2018 [21]

3. Louay Bassbouss, Stephan Steglich, and Sascha Braun. „Towards a high efficient 360° video processing and streaming solution in a multiscreen environment". In: *2017 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*. 2017, pp. 417–422 [20]

4. Louay Bassbouss, Stefan Pham, Stephan Steglich, and Martin Lasak. „Content Preparation and Cross-Device Delivery of 360° Video with 4K Field of View using DASH". in: *2017 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*. Hong Kong, 2017 [173]

5. Louay Bassbouss, Stephan Steglich, and Christian Fuhrhop. „Smart TV 360“. In: *Broadcast Engineering and Information Technology Conference, Virtual and Augmented Reality/Immersive Content*. Las Vegas, USA, 2017 [151]

6. Paul Murdock, Louay Bassbouss, Martin Bauer, Mahdi Ben Alaya, Rajdeep Bhowmik, Rabindra Chakraborty, Mohammed Dadas, John Davies, Wael Diab, Khalil Drira, Bryant Eastham, Charbel El Kaed, Omar Elloumi, Marc Girod-Genet, Nathalie Hernandez, Michael Hoffmeister, Jaime Jiménez, Soumya Kanti Datta, Imran Khan, Dongjoo Kim, Andreas Kraft, Oleg Logvinov, Terry Longstreth, Patricia Martigne, Catalina Mladin, Thierry Monteil, Paul Murdock, Philippé Nappey, Dave Raggett, Jasper Roes, Martin Serrano, Nicolas Seydoux, Eric Simmon, Ravi Subramaniam, Joerg Swetina, Mark Underwood, Chonggang Wang, Cliff Whitehead, and Yongjing Zhang. *Semantic interoperability for the Web of Things*. White Paper. Online: `http://dx.doi.org/10.13140/RG.2.2.25758.13122`. IEEE Standards Association, AIOTI, oneM2M and W3C Joint Collaboration, Aug. 2016 [174]

7. Louay Bassbouss, Stephan Steglich, and Martin Lasak. „Best Paper Award: High Quality 360° Video Rendering and Streaming“. In: *Media and ICT for the Creative Industries*. Porto, Portugal, 2016 [19]

8. Louay Bassbouss and Stephan Steglich. „Position Paper: High quality 360° Video Rendering and Streaming on the Web“. In: *W3C Workshop on Web and Virtual Reality*. San Jose, CA, USA, 2016 [175]

9. Louay Bassbouss. *Einführung in das Physical Web*. Electronic Document. Online: `https://heise.de/-2919078`. Heise Developer, 2015 [176]

10. Louay Bassbouss, Görkem Güçlü, and Stephan Steglich. „Towards a remote launch mechanism of TV companion applications using iBeacon“. In: *2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE)*. Tokyo, Japan, 2014, pp. 538–539 [177]

11. Louay Bassbouss, Görkem Güçlü, and Stephan Steglich. „Towards a wake-up and synchronization mechanism for Multiscreen applications using iBeacon“. In: *2014 International Conference on Signal Processing and Multimedia Applications (SIGMAP)*. Vienna, Austria, 2014, pp. 67–72 [18]

12. Christopher Krauss, Louay Bassbouss, Stefan Pham, Stefan Kaiser, Stefan Arbanowski, and Stephan Steglich. „Position Paper: Challenges for enabling targeted multi-screen advertisement for interactive TV services“. In: *W3C Web and TV Workshop*. Munich, Germany, 2014 [178]

13. Louay Bassbouss, Max Tritschler, Stephan Steglich, Kiyoshi Tanaka, and Yasuhiko Miyazaki. „Towards a Multi-screen Application Model for the Web“. In: *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*. Kyoto, Japan, 2013, pp. 528–533 [17]

14. Jean-Claude Dufourd, Louay Bassbouss, Max Tritschler, Radhouane Bouazizi, and Stephan Steglich. „An Open Platform for Multiscreen Services“. In: *EuroITV 2013: 11th European Interactive TV Conference*. Como, Italy, 2013 [179]

15. Evanela Lapi, Nikolay Tcholtchev, Louay Bassbouss, Florian Marienfeld, and Ina Schieferdecker. „Identification and Utilization of Components for a Linked Open Data Platform“. In: *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*. Izmir, Turkey, 2012, pp. 112–115 [180]

16. Robert Kleinfeld, Louay Bassbouss, Iosif Alvertis, and George Gionis. „Empowering Civic Participation in the Policy Making Process through Social Media“. In: *International AAAI Conference on Web and Social Media*. Dublin, Ireland, 2012 [181]

17. George Gionis, Louay Bassbouss, Heïko Desruelle, Dieter Blomme, John Lyle, and Shamal Faily. „'Do we know each other or is it just our devices?': a federated context model for describing social activity across devices“. eng. In: *Federated Social Web Europe 2011, Proceedings*. Berlin, Germany: W3C ; PrimeLife, 2011, p. 6 [182]

18. Heiko Pfeffer, Louay Bassbouss, David Linner, Françoise Baude, Virginie Legrand, Ludovic Henrio, and Paul Naoumenko. „Mixing Workflows and Components to Support Evolving Services“. In: *International Journal of Adaptive, Resilient and Autonomic Systems* 1.4 (2010), pp. 60–84 [183]

19. Iacopo Carreras, Louay Bassbouss, David Linner, Heiko Pfeffer, Vilmos Simon, Endre Varga, Daniel Schreckling, Jyrki Huusko, and Helena Rivas. „BIONETS: Self Evolving Services in Opportunistic Networking Environments“. In: *Bioinspired Models of Network, Information, and Computing Systems: 4th International Conference, BIONETICS 2009, Avignon, France, December 9-11, 2009*. Ed. by Eitan Altman, Iacopo Carrera, Rachid El-Azouzi, Emma Hart, and Yezekael Hayel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 88–94 [184]

20. Heiko Pfeffer, Louay Bassbouss, and Stephan Steglich. „Structured Service Composition Execution for Mobile Web Applications“. In: *2008 12th IEEE*

International Workshop on Future Trends of Distributed Computing Systems. Kunming, China, Oct. 2008, pp. 112–118 [125]

A.2 Patents

1. Martin Lasak, Louay Bassbouss, and Stephan Steglich. *[DE] Verarbeitungsverfahren und Verarbeitungssystem für Videodaten*. Patent. Patent Number: DE 102017125544B3, Published June 28, 2018, Online: `https://depatisnet.dpma.de/DepatisNet/depatisnet?action=bibdat&docid=DE102017125544B3`. June 2018 [164]

2. Martin Lasak, Louay Bassbouss, and Stephan Steglich. *Processing Method and Processing System for Video Data*. Patent. Patent Number: WO2018210485, Published November 22, 2018, Online: `https://patentscope.wipo.int/search/en/detail.jsf?docId=WO2018210485`. Nov. 2018 [165]

A.3 Contribution to Standards

1. W3C. *Second Screen Working Group*. Tech. rep. Online: `https://www.w3.org/2014/secondscreen/`. The World Wide Web Consortium (W3C), 2017 [12]

2. *Open Screen Protocol*. Open Source Specification. Online: `https://github.com/webscreens/openscreenprotocol`. The World Wide Web Consortium (W3C), 2017 [16]

3. W3C. *Media and Entertainment Interest Group*. Tech. rep. Online: `https://www.w3.org/2011/webtv/`. The World Wide Web Consortium (W3C), 2017 [185]

4. W3C. *Web of Things*. Tech. rep. Online: `https://www.w3.org/WoT/`. The World Wide Web Consortium (W3C), 2017 [186]

5. *HbbTV 2.0.1 Specification, Companion Screen and Media Synchronization Sections*. Tech. rep. Online: `http://www.etsi.org/deliver/etsi_ts/102700_102799/102796/01.04.01_60/ts_102796v010401p.pdf`. Hybrid broadcast broadband TV (HbbTV), 2016 [42]

A.4 Supervision Support of Theses

1. Igor Fritzsch. „360° Storytelling:Mixed Media, Analytics and Interaction Design“. MA thesis. Technical University of Berlin, June 2019 [187]

2. Thomas Fett. „Design and Implementation of a Sound Engine for 360° Videos in Web Browsers and Smart TVs“. MA thesis. Technical University of Berlin, Dec. 2018 [188]

3. Christian Bach. „360° Video Streaming for Head-Mounted Displays“. MA thesis. Wildau Technical University of Applied Science, June 2018 [189]

4. Marius Wessel. „Assembler on the Web - Evaluation of the WebAssembly Technology“. MA thesis. Technical University of Berlin, June 2018 [190]

5. Lukas Rögner. „Cloud-Based Application Rendering for Low-Capability Devices“. MA thesis. Technical University of Berlin, Oct. 2017 [191]

6. Christian Bromann. „Design and Implementation of a Development and Test Automation Platform for HbbTV“. MA thesis. Technical University of Berlin, Aug. 2017 [192]

7. Jonas Rook. „Konzipierung und Entwicklung eines W3C konformen Web of Things Framework“. MA thesis. HTW Berlin, Mar. 2017 [193]

8. Akshay Akshay. „Analysis and Implementation of Unified Synchronization Framework for HbbTV2.0 Sync-API and W3C Web-Timing API“. MA thesis. Kiel University of Applied Sciences, Feb. 2016 [194]

9. Tommy Weidt. „Synchronization Framework for W3C Second Screen Presentation API“. MA thesis. Technical University of Berlin, June 2015 [195]

10. Yi Fan. „Platform for sharing and synchronization of web content in multiscreen applications“. MA thesis. Technical University of Berlin, Jan. 2015 [196]

11. Kostiantyn Kahanskyi. „Dynamic Media Objects“. MA thesis. Technical University of Berlin, Apr. 2014 [197]

12. Anne Haase. „Design and implementation of a migration framework for multiscreen applications“. MA thesis. Free University of Berlin, Jan. 2014 [198]

13. Lutz Welpelo. „Plattform zur Verfolgung von Produkt- und Markenpiraterie auf Online-Marktplätzen“. MA thesis. Technical University of Berlin, July 2013 [199]

14. Alexander Futasz. „Web Scraping Cloud Platform With Integrated Visual Editor and Runtime Environment“. MA thesis. Technical University of Berlin, Mar. 2013 [200]

15. Michal Radziwonowicz. „Development and Cross-domain Runtime Environment for Distributed Mashups“. MA thesis. Technical University of Berlin, Jan. 2013 [201]

16. Ahmad Abbas. „Cloud Platform for Web Connected Sensors and Actuators“. MA thesis. Beuth University of Applied Sciences Berlin, Dec. 2012 [202]

17. Niklas Schmücker. „Enhancing Web-Based Citizen Reporting Platforms for the Public Sector through Social Media“. MA thesis. Technical University of Berlin, Feb. 2012 [203]

18. Hui Deng. „Click-By-Click Mashup Platform for Open Statistical Data“. MA thesis. Technical University of Berlin, Apr. 2011 [204]

19. Alexander Kong. „Securing Semi-automatic Data flow Control in Government Mashups“. MA thesis. Technical University of Berlin, Dec. 2010 [205]

20. Jie Lu. „Towards an End-User Centric Mashup Creation Environment facilitated through Code Sharing“. MA thesis. Technical University of Berlin, June 2010 [206]

A.5 Open Source Contributions

1. *peer-ssdp: Node.js Implementation of the Simple Service Discovery Protocol SSDP*. Open Source Implementation. Online: `https://github.com/fraunhoferfokus/peer-ssdp/`. Fraunhofer FOKUS, 2017 [140]

2. *peer-upnp: Node.js Implementation of the Universal Plug and Play Protocol UPnP*. Open Source Implementation. Online: `https://github.com/fraunhoferfokus/peer-upnp/`. Fraunhofer FOKUS, 2017 [207]

3. *peer-dial: Node.js Implementation of the Discovery and Launch Protocol DIAL*. Open Source Implementation. Online: `https://github.com/fraunhoferfokus/peer-dial/`. Fraunhofer FOKUS, 2017 [208]

4. *node-hbbtv: Node.js Implementation of the HbbTV Companion Screen Specification*. Open Source Implementation. Online: `https://github.com/fraunhoferfokus/node-hbbtv/`. Fraunhofer FOKUS, 2017 [209]

5. *cordova-plugin-hbbtv: Cordova Plugin Implemenation of the HbbTV Companion Screen Specification*. Open Source Implementation. Online: `https://github.com/fraunhoferfokus/cordova-plugin-hbbtv`. Fraunhofer FOKUS, 2017 [141]

6. *cordova-plugin-presentation: Cordova Plugin Implemenation of the W3C Second Screen Presentation API for Airplay and Miracast*. Open Source Implementation. Online: `https://github.com/fraunhoferfokus/cordova-plugin-presentation`. Fraunhofer FOKUS, 2017 [210]

7. *Concept and Implementation of UPnP/SSDP Support in Physical Web*. Open Source Implementation. Online: `https://github.com/google/physical-web/blob/master/documentation/ssdp_support.md`. Google, 2017 [211]

A.6 University Courses And Guest Lectures

1. Louay Bassbouss and Christopher Krauß. *Personalized Multi-Platform Development*. Guest Lecture. Beuth University of Applied Sciences Berlin, Feb. 2015 [212]

2. Louay Bassbouss. *Multiscreen Technologies, Standards and Best Practices*. Guest Lecture. Beuth University of Applied Sciences Berlin, May 2015 [213]

3. Louay Bassbouss. *Multiscreen Technologies and Standards*. Guest Lecture. Beuth University of Applied Sciences Berlin, Jan. 2017 [214]

4. Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Lecture WS 2015/2016*. University Course. Technical University Berlin, 2015 [215]

5. Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Project SS 2016*. University Course. Technical University Berlin, 2016 [216]

6. Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Lecture WS 2016/2017*. University Course. Technical University Berlin, 2016/17 [217]

7. Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Project WS 2016/2017*. University Course. Technical University Berlin, 2016/17 [218]

8. Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Project SS 2017*. University Course. Technical University Berlin, 2017 [219]

9. Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Lecture WS 2017/2018*. University Course. Technical University Berlin, 2017/18 [220]

10. Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Project WS 2017/2018*. University Course. Technical University Berlin, 2017/18 [221]

11. Stephan Steglich, Louay Bassbouss, Stefan Pham, Christopher Krauß, and Andre Paul. *Advanced Web Technologies Project SS 2018*. University Course. Technical University Berlin, 2018 [222]

B Multiscreen Web Application Examples

This chapter provides the source code for two multiscreen applications used as examples in this thesis. Section B.1 covers the implementation for the components of the "*Multiscreen Slides Application*", while Section covers the *Multiscreen Application Tree* and the implementation for the component of the "*Video Wall Multiscreen Application*".

B.1 Multiscreen Slides Application

```
<template id="aac-control">
 <style>
  /* styles for the control AAC */
 </style>
 <div>
  <button id="open-btn">Open Slides</button><br>
  <button id="prev-btn">Previous Slide</button><br>
  <button id="next-btn">Next Slide</button><br>
 </div>
</template>

<script>
 class AACControl extends AAC {
  connectedCallback() {
   var template = document.querySelector('#aac-control').content;
   var shadow = this.attachShadow({mode: 'open'});
   shadow.appendChild(document.importNode(template, true));
   var openBtn = shadow.querySelector("#open-btn");
   var prevBtn = shadow.querySelector("#prev-btn");
   var nextBtn = shadow.querySelector("#next-btn");
   var state = this.msa.object("state",{
    currSlide: 0,
    slides: []
   });
   openBtn.onclick = function(){
    this.loadSlides().then(function(slides){
     state.slides = slides;
    });
   }
   prevBtn.onclick = function(){
```

```
    state.currSlide > 0 && state.currSlide--;
   }
   nextBtn.onclick = function(){
    state.currSlide < slides.length-1 && state.currSlide++;
   }
  }
  loadSlides(){
   /* load slides from somewhere */
  }
 }
 customElements.define('aac-control', AACControl);
</script>
```

Listing B.1: Control Atomic Component

```
<template id="aac-preview">
 <style>
  /* styles for the preview AAC */
 </style>
 <div>
  <p id="preview"></p>
 </div>
</template>

<script>
 class AACPreview extends AAC {
  connectedCallback() {
   var template = document.querySelector('#aac-preview').content;
   var shadow = this.attachShadow({mode: 'open'});
   shadow.appendChild(document.importNode(template, true));
   var previewEl = shadow.querySelector("#preview");
   var state = this.msa.object("state",{
    currSlide: 0,
    slides: []
   });
   state.observe("*","change",function(path, newVal, oldVal){
    var slide = state.slides[state.currSlide];
    previewEl.innerHTML = slide && slide.content? slide.content:"";
   });
  }
 }
 customElements.define('aac-preview', AACPreview);
</script>
```

Listing B.2: Preview Atomic Component

```
<template id="aac-notes">
 <style>
  /* styles for the notes AAC */
 </style>
 <div>
  <p id="notes"></p>
```

```
 </div>
</template>

<script>
 class AACNotes extends AAC {
  connectedCallback() {
   var template = document.querySelector('#aac-notes').content;
   var shadow = this.attachShadow({mode: 'open'});
   shadow.appendChild(document.importNode(template, true));
   var notesEl = shadow.querySelector("#notes");
   var state = this.msa.object("state",{
    currSlide: 0,
    slides: []
   });
   state.observe("*","change",function(path, newVal, oldVal){
    var slide = state.slides[state.currSlide];
    notesEl.innerHTML = slide && slide.notes? slide.notes:"";
   });

  }
 }
 customElements.define('aac-notes', AACNotes);
</script>
```

Listing B.3: Notes Atomic Component

```
<template id="cac-presenter">
 <style>
  /* styles for the presenter CAC */
 </style>
 <div>
  <button id="present-btn">Present</button>
  <aac-preview></aac-preview>
  <aac-notes></aac-notes>
  <aac-control></aac-control>
 </div>
</template>

<script>
 class CACPresenter extends CAC {
  connectedCallback() {
   var template = document.querySelector('#cac-presenter').content;
   var shadow = this.attachShadow({mode: 'open'});
   shadow.appendChild(document.importNode(template, true));
   var presentBtn = shadow.querySelector("#present-btn");
   var self = this;
   presentBtn.onclick = function(){
    self.discoverFirstDevice().then(function(device){
     device.launch("cac-display");
    }).catch(function(err){
     /* no device found */
    });
```

```
    }
   }
   discoverFirstDevice() {
    var self = this;
    return new Promise(function(resolve, reject){
     self.ondevicefound = function(evt){
      self.stopDiscovery();
      resolve(evt.device);
     }
     setTimeout(function(){
      self.stopDiscovery();
      reject(new Error("No device found"));
     },5000);
    });
   }
 }
 customElements.define('cac-presenter', CACPresenter);
</script>
```

Listing B.4: Presenter Composite Component

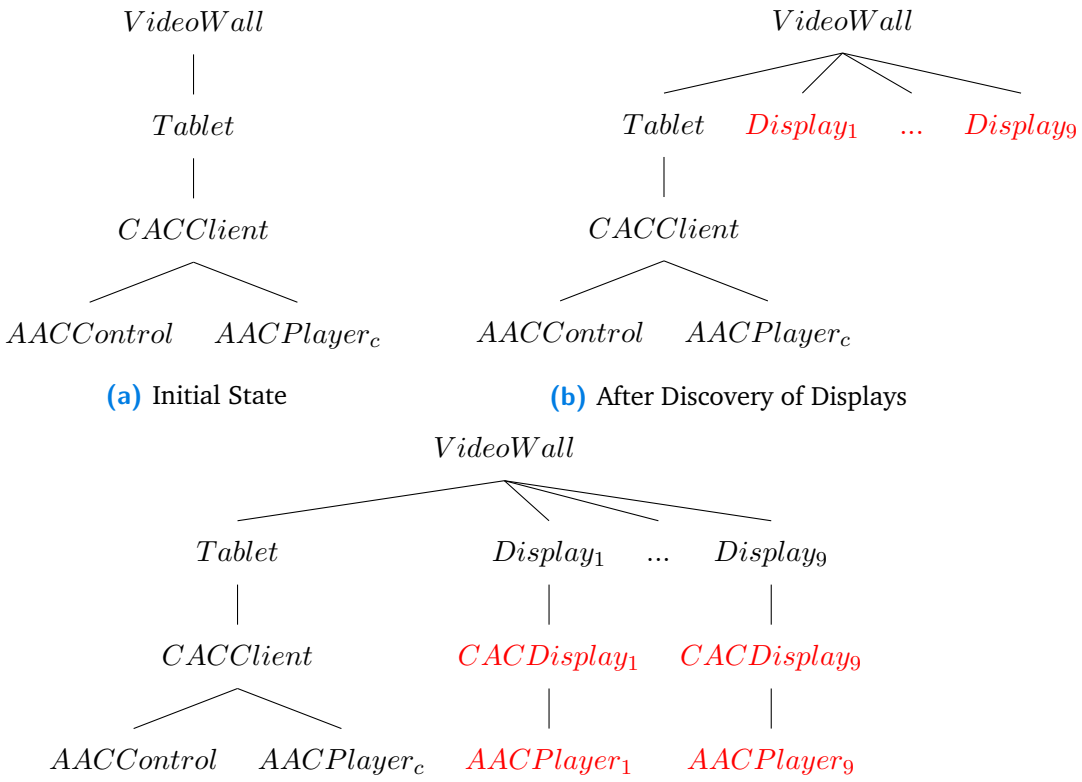
```
<template id="cac-display">
 <style>
  /* styles for the display CAC */
 </style>
 <div>
  <aac-preview></aac-preview>
 </div>
</template>

<script>
 class CACDisplay extends CAC {
  connectedCallback() {
   var template = document.querySelector('#cac-display').content;
   var shadow = this.attachShadow({mode: 'open'});
   shadow.appendChild(document.importNode(template, true));
  }
 }
 customElements.define('cac-display', CACDisplay);
</script>
```

Listing B.5: Display Composite Component

B.2 Video Wall Multiscreen Application

B.2.1 Multiscreen Application Tree



(a) Initial State

(b) After Discovery of Displays

(c) After Launch of Display CAC

Figure B.1.: Video Wall Multiscreen Application Tree

B.2.2 Implementation

```
<template id="aac-control">
 <style>
  /* styles for the control AAC */
 </style>
 <div>
  <button id="present-btn">Present</button>
  <dialog>
   <ul id="display-list"></ul>
   <button id="launch-btn">Launch</button>
   <button id="close-btn">Close</button>
  </dialog>
 </div>
</template>

<script>
```

```
class AACControl extends AAC {
 connectedCallback() {
  var template = document.querySelector('#aac-control').content;
  var shadow = this.attachShadow({mode: 'open'});
  shadow.appendChild(document.importNode(template, true));
  var dialog = shadow.querySelector("dialog");
  var presentBtn = shadow.querySelector("#present-btn");
  var launchBtn = shadow.querySelector("#launch-btn");
  var closeBtn = shadow.querySelector("#close-btn");
  var displays = {};
  var msa = this.msa;
  var cacClient = this.cac;
  msa.ondevicefound = function(evt){
var display = evt.device;
   displays[display.id] = display;
   //update display list in the UI
  };

  msa.ondevicelost = function(evt){
var display = evt.device;
delete displays[display.id];
   //update display list in the UI
  };

  dialog.onopen = function(){
   msa.startDiscovery();
  };

  dialog.onclose = function(){
   msa.stopDiscovery();
displays = {};
// empty display list in the UI
  };

  presentBtn.onclick = function(){
dialog.showModal();
  };

  launchBtn.onclick = function(){
displays.forEach(function(display){
 display.connect().then(function(){
     return display.addCAC("cac-display");
 }).then(function(cacDisplay){
  return cacDisplay.getAAC("aac-player");
 }).then(function(aac){
  var tileUrl = getVideoUrl(display.name);
  aac.postMessage(tileUrl);
 });
});
   var videoUrl = getVideoUrl();
   var aacPlayer = cacClient.getAAC("aac-player");
```

```
 aacPlayer.postMessage(videoUrl);
 dialog.closeModal();
   };

   closeBtn.onclick = function(){
 msa.devices.forEach(function(display){
     // only disconnect() will not close the CACDisplay
     display.removeCAC("cac-display").then(function(){
      display.disconnect();
  });
 });
   };
  }

  getVideoUrl(displayName){
   // returns video tile URL of corresponding display
   // returns video URL for client if no input porovided
  }
 }
 customElements.define('aac-control', AACControl);
</script>
```

Listing B.6: Control Atomic Component

```
<template id="aac-player">
 <div>
  <video id="video"></video>
 </div>
</template>

<script>
 class AACPlayer extends AAC {
  connectedCallback() {
   var template = document.querySelector('#aac-player').content;
   var shadow = this.attachShadow({mode: 'open'});
   shadow.appendChild(document.importNode(template, true));
   var video = shadow.querySelector("#video");
   var syncGroup = this.msa.syncGroup("VideoWall");
   syncGroup.addMedia(video);
   // call syncGroup.removeMedia(video); to end synchronization
  }
 }
 customElements.define('aac-player', AACPlayer);
</script>
```

Listing B.7: Preview Atomic Component

```
<template id="cac-client">
 <div>
  <aac-player></aac-player>
  <aac-control></aac-control>
 </div>
```

```
</template>

<script>
 class CACClient extends CAC {
  connectedCallback() {
   var template = document.querySelector('#cac-client').content;
   var shadow = this.attachShadow({mode: 'open'});
   shadow.appendChild(document.importNode(template, true));
  }
 }
 customElements.define('cac-client', CACClient);
</script>
```

Listing B.8: Control Atomic Component

```
<template id="cac-display">
 <div>
  <aac-player></aac-player>
 </div>
</template>

<script>
 class CACDisplay extends CAC {
  connectedCallback() {
   var template = document.querySelector('#cac-display').content;
   var shadow = this.attachShadow({mode: 'open'});
   shadow.appendChild(document.importNode(template, true));
  }
 }
 customElements.define('cac-display', CACDisplay);
</script>
```

Listing B.9: Preview Atomic Component

Declaration

I hereby declare in lieu of an oath that I have produced this work by myself. All used sources are listed in the bibliography and content taken directly or indirectly from other sources is marked as such. This work has not been submitted to any other board of examiners and has not yet been published.

Berlin, September 12, 2019

Dipl.-Ing. Louay Bassbouss