NTNU – Trondheim
Norwegian University of
Science and Technology

Cross-Device Application Mobility

Investigating and showcasing the feasibility of
a generic, cross-Device Session Mobility
Platform for Applications

Øystein Wethe Hanssen
Dimitrios Tsigouris

Master of Science in Communication Technology
Submission date: June 2013
Supervisor: Peter Herrmann, ITEM
Co-supervisor: Humberto Castejon, Telenor

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Name of students: Dimitrios Tsigouris and Øystein Wethe Hanssen

More and more devices, such as smartphones, smart TVs and tablets, are nowadays able to run applications. Unfortunately, the applications written for one device are not always compatible with other devices, nor does it always allow you to move a running application from one device to another, while resuming the session on the target device from where you left off on the source device.

Lately, a number of solutions have been proposed for service session handover across devices. These solutions are in most cases service specific. The aim of this project is to find generic mechanisms for service session handover across devices. We anticipate that a universal solution, applicable to all types of services, will most probably not be feasible, and envision a solution combining both a generic part, and a service-specific part, which will be different for each (type of) service. The project will propose such a solution. For this, existing commercial and academic solutions for different kind of services will be analyzed, as well as the real needs of end-users.

To validate the proposed solution, a proof of concept will be implemented for a selected service/app.

Assignment given: 17. Cross-device mobility for HTML5 applications

Supervisor: Humberto Castejón, Telenor Research and Future Studies

Abstract

Today, cross-device capabilities has become the de facto standard among most applications, enabling users to access an application, and in some cases, resume their previous session, from a various of devices. While this allows the user to run the same application on different devices, there still exists no widespread solution providing the ability to transfer an ongoing application session from one device to another, continuing the ongoing session immediately. As users we've asked have considered such functionality to be of potential great value, this is definitely something worth exploring. In this thesis, we have investigated and proved that a generic, cross-device session mobility platform is possible and feasible, but not without a certain level of application modification, which we believe is a reasonable requirement.

A truly generic cross-device session mobility platform should work with any application, irrespective of the type of the application and the type of terminal it is running on. This becomes a challenge, considering that the nature of an application may vary a lot from application to application.

Our proposed solution, the Migration Platform (MP), is realized as a centralized peer-to-peer architecture. It consists of two entities; the Migration Server (MS) and the Migration Client (MC), the latter installed as a software on each device. Together, the MS and MC will provide any application with cross-device session mobility capabilities.

Upon switching device (i.e. upon a migration), a direct data channel between source and target device is established. The state of the application about to be migrated is transferred over this data channel, and passed to the application on the target device. This enables the application to resume the application session from where it was left off on the source device. By suggesting a workaround, we also make this scheme viable for real-time communication applications.

The applications that are to utilize the capabilities provided by the MP need to implement an interface to the MC, as well as a user interface enabling the user to trigger migration functions from the application. This way, the application can invoke migration functions provided by the MC, and the MC can invoke the application-specific functions specified in the interface. By having the application developers implement the required functions, we provide them with complete freedom when it comes to how they want to apply the migration functionalities, both when it comes to behavior and appearance.

We implemented a proof of concept in the web environment, primarily based on the bleeding edge technology, WebRTC, an API currently being drafted, enabling browser to browser communication. Using the exact same generic scripts, we implement session mobility features in four, arguably different, demo applications; a HTML5 video streaming application, a YouTube streaming application, a videochat application and a

Chrome browsing extension. With this, we showcased the behavior of the session mobility features, as well as proved the viability of our proposed solution.

Sammendrag

I dag er det vanlig med applikasjoner som er tilgjengelige på tvers av enheter. Dette tillater en bruker å aksessere en applikasjon, og, i noen tilfeller, fortsette en tidligere sesjon fra en rekke forskjellige enheter. Selv om dette tillater en bruker å kjøre samme applikasjon på forskjellige enheter, finnes det fortsatt ingen utbredt løsning som tilbyr muligheten til å overføre en pågående applikasjonssesjon fra en enhet til en annen, hvor man fortsetter sesjonen umiddelbart. Da brukerne vi har spurt anser at en slik funksjonalitet vil ha potensielt stor nytteverdi, er det noe som defintivt er verdt å utforske videre. I denne masteroppgaven har vi undersøkt og bevist at en generisk, kryssenhetssesjonsmobilitetsplattform både er mulig og gjennomførbart, men ikke uten en viss grad av applikasjonsmodifikasjon, noe vi mener er rimelig.

En virkelig generisk kryssenhetssesjonsmobilitetsplattform burde fungere med enhver applikasjon, uavhengig av applikasjonstype og/eller hva slags enhet den kjører på. Dette skaper en utfordring, da applikasjoner (og enheter) kan være svært forskjellige av natur.

Vår foreslåtte løsning, Migreringsplattformen (MP), er realisert som en sentralisert peer-to-peer-arkitektur. Den består av to entiteter; Migreringsserveren (MS) og Migreringsklienten (MC), hvor den siste blir installert som programvare på hver enhet. Til sammen skal de gi enhver applikasjon kryssenhetssesjonsmobilitetsevner.

Ved bytting av enheter (i.e. ved en migrering), opprettes det en direkte datakanal mellom kildeenheten og målenheten. En beskrivelse av tilstanden/sesjonen til den kjørende applikasjonen (som er i ferd med å bli migrert) blir overført over denne datakanalen, og blir gitt til applikasjonen på målenheten. Vi har også gjort denne metoden anvendelig for sanntidskommunikasjonsapplikasjoner ved å foreslå en ytterligere, men liten applikasjonsmodifikasjon.

Applikasjonene som skal benytte funksjonalitene tilbudt av MPen må implementere et grensesnitt til MCen, i tillegg til et brukergrensesnitt som gjør at brukerne av applikasjonen kan trigge de ulike migreringsfunksjonene. På denne måten kan applikasjonen kalle på migreringsfunksjonene som tilbys av MCen, samtidig som MCen kan kalle på applikasjonsspesifikke funksjoner spesisifert via grensesnittet. Ved å kreve at applikasjonsutviklerne må implementere noen av funksjonene, gir vi dem full frihet når det kommer til hvordan de ønsker å anvende migreringsfunksjonene, både med tanke på oppførsel og utseende.

Vi implementerte et “proof of concept“ i weben, primært basert på WebRTC, et API under utarbeidelse som muliggjør direkte kommunikasjon mellom nettlesere. Ved å bruke de samme generiske scriptene, implementerte vi sesjonsmobilitet i fire relativt forskjellige demowebapplikasjoner; en HTML5-videostrømmingsapplikasjon, en YouTube-videostrømmingsapplikasjon, en videochatapplikasjon, og en nettlesingsutvidelse. Med

dette har vi vist hvordan de ulike migeringsfunksjonene kan fungere, samt bevist at vår foreslåtte løsning både er anvendelig og levedyktig.

Preface

The purpose of this report is to document and present the process of our work, in the field of cross-device session mobility. The project that we have undertaken was proposed by the Department of Telematics at NTNU, and by Telenor, as part of NTNU's Master's thesis suggestion list for Spring 2013. Originally, the proposed project was a study of cross-device mobility within the limits of HTML5. When we undertook the project in January 2013, we decided to widen the project's focus to a more generic study of cross-device session mobility.

Our report consists of a theoretical analysis of the state of the art technology and related work within the field of session mobility, a theoretical approach in our effort to design and propose a solution, and finally an implementation of the essential parts of our design, as a proof of concept. We feel that we have provided a sufficient overview of the existing technologies and solutions, and have identified what is and what is not applicable for a generic cross-device session mobility solution. Our solution design, as well as our proof of concept, has certain limitations, but we feel that we have surpassed the capabilities of other existing session mobility solutions, which we are very pleased with.

In our effort, we got significant help from Humberto Martinez Castejón and Frode Kileng from Telenor Research and Future Studies, who helped us a lot with their insight and knowledge, both on our theoretical and practical approaches.

Contents

List of figures

List of tables

List of acronyms

API	Application Programming Interface
AP	Access Point
CIO	Chief Information Officer
CC	Cloud Computing
CSS	Cascading Style Sheets
DC	(WebRTC) Data Channel
DIAL	DIscover And Launch
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
ICE	Interactive Connectivity Establishment
IM	Instant Messaging
IP	Internet protocol
JS	JavaScript
JSEP	Javascript Session Establishment Protocol
JSON	JavaScript Object Notation
LAN	Local area network
NAT	Network Address Translation
OS	Operating System
P2P	Peer-to-Peer
PC	(WebRTC) Peer Connection
SDK	Software Development Kit
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SOTA	State of the Art
SRTP	Secure Real-time Transfer Protocol
SSDP	Simple Service Discovery Protocol
STUN	Session Traversal Utilities for NAT

TCP	Transmission Control Protocol
TURN	Traversal Using Relays around NAT
UI	User interface
UPnP	Universal Plug and Play
WebRTC	Web Real-Time Communication
XML	Extensible Markup Language

1 Introduction

1.1 Problem and motivation

Today, it is common to see applications being available across most, if not all, types of devices, such as smartphones, tablets, computers, video game consoles and smart TVs. Many of these applications also provide the opportunity to resume a session from where you left off the last time you ran the application. Such functionalities are becoming the de facto standard across a broad range of application categories, be it file storage, document editing, web browsing, multimedia or communication applications. It is a result of the ubiquitous nature of Internet connectivity, and is realized by utilizing technologies such as Cloud Computing, and cross-platform development and user interface tools such as HTML5.

While this allows you to run the same application on different devices, only a few applications offer the ability to transfer an ongoing application from one device to another, continuing the ongoing session immediately. This type of mobility we denote as session mobility, and it will be the main subject of research in this thesis. Due to the ever-increasing cross-device availability, such functionality is of potential great utility, but there is not yet any widespread solution providing this. Thus, in this project we investigated the feasibility of designing a cross-device session mobility platform.

Before investigating the feasibility, however, we need to check if such a solution already exists. There has been, and currently is, a lot of research in the field of session mobility. A number of solutions have been proposed, though these solutions are in most cases service specific. But a few try to find a more generic, universally applicable solution. We will present to you the various proposed approaches, and discuss and compare their advantages and disadvantages.

1.2 Goals and methodology

Our main goal is threefold:

1. To investigate and determine the feasibility of a generic, cross-device session mobility platform
2. To find and propose a solution to the challenge
3. To showcase the viability of the solution design by implementing a proof of concept, along with a couple of demo applications

Before starting to design a solution, we need to determine whether a generic session mobility platform is feasible or not, and if so, identify the application categories that can benefit from session mobility.

In order to derive use cases and requirements, we will conduct a user study to identify the wants and needs of the actual end user. There, we ask the end users what would be considered useful by them, and as such derive a general idea of which functionalities are seen as attractive, and which are not.

Based on our findings, both from existing solutions and related work, as well as the derived use cases and requirements, we can start discuss and finally develop the design of our own solution.

With the proposed solution design as a backbone, we can choose a development environment on which to implement a proof of concept, as well as demo applications, showcasing the viability of the proposed solution.

1.3 Outline

In Chapter 2, we will introduce the reader to useful definitions and terms used throughout the report, we will also explore what a generic session mobility platform actually is, and what it entails. Here, we will also identify the application categories that can benefit from session mobility.

In Chapter 3 we will present state-of-the-art technologies frequently used within each of the application categories we have identified in Chapter 2. We will also see how they are applied to provide the most popular cross-device applications.

In Chapter 4 we will look at the related work, as well as existing solution, within the field of session mobility, which we will later use for inspiration when discussing our design.

Chapter 5 presents the results from our user study, which are used to derive use cases and finally requirements for our solution.

Chapters 6 and 7 are used to discuss and present the design of our solution, respectively, using the results and information gathered from the previous chapters, while Chapter 8 presents our implemented proof of concept.

Finally, we conclude our work and summarize our results in Chapter 9, before we discuss any further work in Chapter 10.

2 Background

2.1 Introduction

In this chapter we will introduce useful definitions and terms that we will use throughout this paper. These definitions and terms are in the context of a cross-device session mobility platform, i.e. some type of platform providing the ability to transfer an ongoing application session from one device to another.

We will also explore what a generic session mobility platform entails, and introduce the application categories we've identified, as well as discuss application type implications.

2.2 Useful definitions/terms

- Session Mobility

The transfer of an ongoing session from one device to another. (1)

- Cross-device:

Referring to an application, or a service, available on multiple devices, e.g. smartphones, tablets, computers, smart TVs, set top boxes and/or video game consoles.

- Terminal mobility:

Allowing a device to change location (i.e. change its access point) and still be able to communicate, i.e. without terminating the ongoing session. (2)

- Personal Mobility:

The ability to access a given service the same way from different terminals. (3)

- Cross-domain:

In which two clients located in different security domains are able to communicate and send/receive data to each other.

- Migration:

The process of moving something from one place to another. In this paper, this is referring to the process of moving an application session's state from one device to another.

- State:

Describes the ongoing system or entity session at a given instant in time.

2.3 Exploring a generic session mobility platform

When we consider a generic session mobility platform for applications, we refer to a solution that should work with *any* type of application on *any* type of terminal/device. That is, it should be a service/platform providing session mobility irrespective of both the terminal/device (i.e. cross-device) and the application (i.e. generic). See Figure 1.

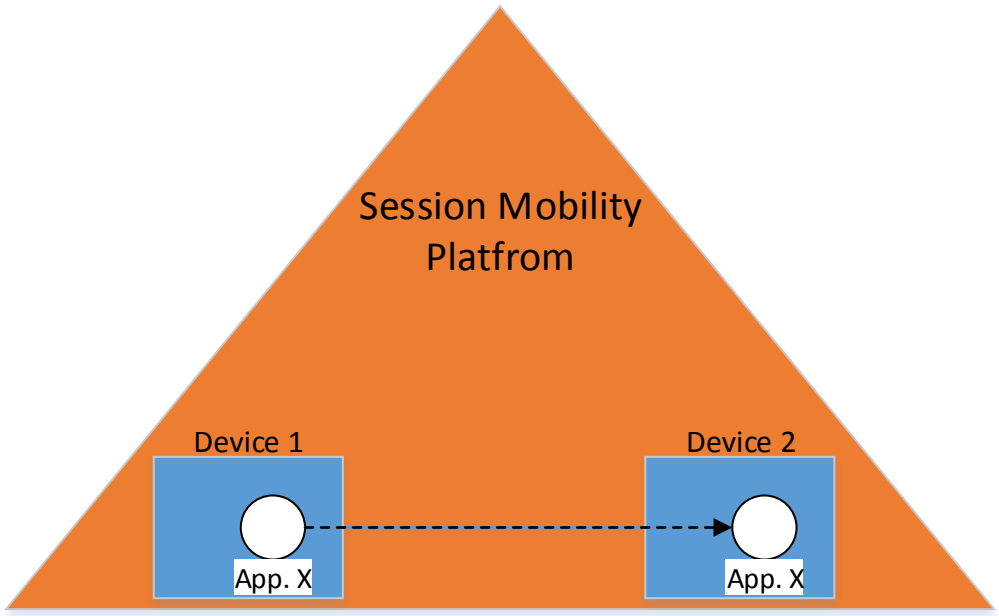

Figure 1. A platform providing generic, cross-device session mobility for applications.

In later chapters we will discover there are several potential ways to create such a platform, as well as several ways to realize session mobility. We will examine and discuss in depth these various possibilities. For now it is sufficient to know the concept of a cross-device session mobility platform.

2.3.1 Application categories

Although a generic solution should be able work with any type of application, we need to consider and understand the different types of applications, as they can be very different in nature. Thus, we find it necessary to classify applications into different categories based on their functionality. Inspired by the application categories in Google's Play (4) and Apple's App Store (5), we ended up with the following 5 (aggregated) application categories.

1. **Browsing applications:**

This category refers to all applications based on retrieving/pulling and presenting information from the World Wide Web. This can be anything from a web-site to e.g. a weather forecast app.

2. **Business applications:**

In this category, E-mail, document viewing and editing applications are included.

3. **Multimedia applications:**

This category includes all applications for mostly viewing and sometimes modifying multimedia, such as photos, videos and audio clips. The multimedia being viewed/played can be either a file residing locally on the device viewing it or streamed/downloaded from a remote server.

4. **Communication applications:**

This category includes all applications used for communication between users, such as chat, voice and video calling.

5. **Gaming applications:**

In this category video gaming applications are included.

2.3.2 Application type implications

Since we are considering a generic session mobility platform that should be able to work with any given application regardless of type or real-time requirements, it stands to reason to consider the possible implications this might have.

Real-time vs. non-real-time consideration

We need to take into account the differences between real-time and non-real-time applications. If we are to provide a truly generic solution, the platform has to be able to work with both real-time and non-real-time applications, despite their differences. Can a session transfer be handled in the same way on both types of applications, or do they need to be differentiated?

In *real-time* applications, the information exchanged is “live”. I.e. information is sent immediately upon capture from the input device (e.g. web camera and microphone), directly to the receivers, satisfying strict real-time time constraints. The real-time requirement is most likely to be found in the context of live streaming and communication applications.

Consider e.g. a video chat between two persons. Here, there are strict time constraints that need to be met in order for the persons to be able to communicate properly.

Non-real-time applications, however, usually communicate with application *servers*, not remote peers. As such, there is no "live" element, whether that is live-streaming or live-communication. It is usually based on a client-server paradigm, with no real-time time constraints. Consider e.g. a video streaming application. In this case, buffering of the video can be allowed (although it is desirable to avoid), as the video played is not real-time. Additionally, a user can stop and/or pause a stream at any time he/she pleases, as there is no other person on the other endpoint.

Protocol consideration

Another implication to consider, is that different applications use different technologies (e.g. network, transport and application layer protocols). Additionally, the very same application may use different protocols depending on the device it is running on. To provide a cross-device session mobility platform for any application, we are required to work with all these applications. Thus, the platform has to either be independent of the technology an application uses, or to support all of them.

Architecture consideration

Finally, we need to consider the architecture of the application. While some applications may follow a client-server paradigm, others may be peer-to-peer. Should the application server and/or peers be made aware of a session transfer or not? I.e. we need to consider the other endpoint of an application session, whether it's an application server or a remote peer.

2.4 Conclusion

A truly generic cross-device session mobility platform should work with any application, irrespective of type and the terminal it is running on.

We have classified applications into categories, as well as distinguished between real-time and non-real-time application types. But we have yet to determine whether or not these different types/categories can benefit from session mobility or not. To help determine this, we rely on end user's input, both from a user discussion and a public user-survey, whose findings will be discussed in chapter 5.

Before that, however, we should learn more about the technologies utilized to provide cross-device services, as well as the various ways one can perform session mobility. This is done in chapter 3 and 4, respectively.

3 State-of-the-art cross-device technology

3.1 Introduction

In this chapter we will start by introducing you to two technologies frequently being used by the most popular cross-device application categories we've identified, Cloud Computing and HTML5. We will then see how, or if, these technologies are applied to provide applications with cross-device functionality, by looking at examples of application software within each application category identified in 2.3.1.

3.2 Cloud Computing

In essence, Cloud Computing (CC) is when computing becomes a utility. It has received a lot of attention, and has been one of the top priorities for CIOs worldwide the last few years.

"The cloud" consists of ubiquitous and universally available resources that are pooled and provisioned to the consumers on demand, optimizing resource use, without requiring any human interaction. One of its selling points is its rapid elasticity, or scalability, making a cloud service able to handle a fluctuating amount of requests by monitoring the system and reacting immediately to consumer's demand. Another is the fact that the service becomes universally available, provided you have Internet connectivity. (6)

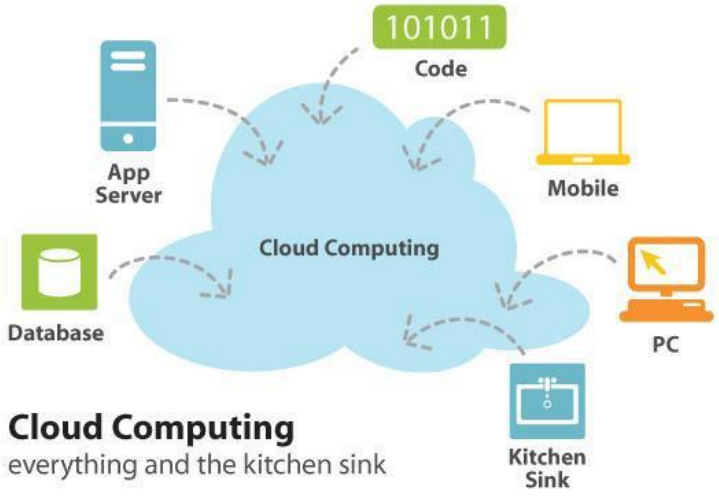

Figure 2. A variety of entities can take part in Cloud Computing. Taken from (7).

With CC, everything is stored and kept up to date in "the cloud", potentially making it accessible from any device at any time. A cloud service is usually accessed via a thin client running on the end user's device (e.g. computer, tablet, smartphone). Commands are sent from this client, but the actual execution (and thus the main workload) are being carried out in the back end – the cloud. This allows both high-end and low-end devices, with e.g. low-performing CPUs and GUPs, to utilize the same cloud services (see Figure 2).

An example of a cloud service is DropBox (8), a personal online storage you can access from a variety of devices, where you can upload and download the files you want. I.e. your files are stored in "the cloud" instead of locally on you device. Another example is Google AppEngine (9), a platform offered by Google where you can both develop and host your own web applications. When deployed, AppEngine helps your application to automatically scale and thus meet the current consumer demand at any time.

In the context of a session mobility platform, we see that CC has met both the cross-device challenge, by having "the cloud" doing the work, as well as partly solved the session mobility challenge, by having the cloud reflecting the current state of your session at any time.

3.3 HTML5

When working with cross-device applications/services, it is important to take into account the various differences of the devices. There needs to exist some sort of device-specific User Interface (UI), ensuring that the interface of the application is properly adapted to the device being currently in use. When focusing on the UI, there are especially two important factors that should be taken into consideration, in order to satisfy an underlying requirement of being user-friendly:

1. The screen size (dimension and resolution)
2. How the user interacts with the device (i.e. the user input)

E.g. there are great differences between a 6" smartphone controlled from a touchscreen, a 15" PC controlled from a mouse and a keyboard, and a 42" TV controlled from a remote control.

HTML5 is the next major revision of the HTML standard, currently under development (though already supported by the major browsers). HTML5's main task is to structure and present content from the WWW. When we talk about HTML5 in this section, we mean not only the markup language HTML5, but also the style sheet language CSS3 and the scripting language JavaScript. Unlike its predecessors, many features of HTML5 "*have been built with the consideration of being able to run on low-powered devices such as*

smartphones and tablets." (10). It allows creating applications that can be run on any device with an HTML5-compatible web browser/renderer (or operating system). According to a recent survey, commissioned by Telerik's Kendo UI, "*the majority of developers now prefer to work with HTML5 instead of native apps for their cross-platform development.*" (11).

One of the major new features (and selling points) of HTML5 is the addition of its new syntactic features, such as the video, audio and the canvas elements. These features are designed to make it easy to enable and handle multimedia and graphical content without having to resort to proprietary plugins and APIs (such as Microsoft Silverlight or Adobe Flash). Additionally, there are several new APIs (10) (some of which was originally part of the HTML5 specification, but now are in separate specifications) that can be used with JavaScript. In this section we will shortly mention a few of them, some of which we will revisit later in this report.

- **Server-sent events.** A technology for providing push notifications from a server to a browser client (12).
- **WebSocket Protocol.** This API enables web pages to take part in a full-duplex (two-way) communication over a single TCP connection (13).
- **WebRTC.** Enables direct browser to browser communication, used for e.g. video/voice chat and peer-to-peer file sharing.
- **Web storage.** Provides behavior similar to cookies, but with larger storage and a better programmatic interface.
- **Offline Web Applications.** Enables a web application to work offline.

Gartner expects HTML5 to make a big impact, as they believe over half of the mobile apps by 2016 will be HTML5 native or hybrid (14). Thus, they urge businesses to prepare for the arrival of HTML5. The importance of HTML5 has already been recognized by application developers, mobile device manufacturers, Telco operators and Internet players, and the adoption of HTML5 is being done with collaboration among the biggest tech players (15).

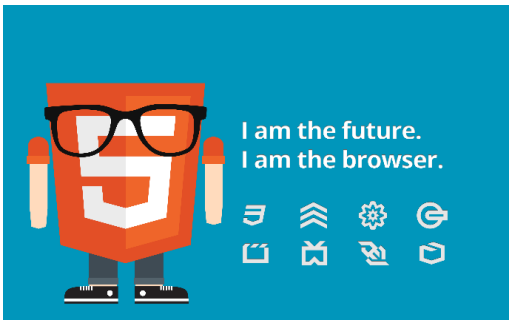

Figure 3. A HTML5-promoting logo taken from w3c (16). The small symbols (from top left) represents HTML5's semantics, CSS3, multimedia, graphics&3D, device access, performance, offline & storage and connectivity selling points.

Currently, there are ongoing projects to develop HTML5-compatible OSs, such as FirefoxOS (17) and Tizen (18). While other mobile operating systems are incorporating HTML5 technology as they go along, Firefox, along with Tizen, are the first to build mobile platforms on HTML5 from the ground up. As such, the HTML5 application is able to run natively on the OS, unlike today, where it is run inside the web browser. "*This means that all of the phone features, including calling, messaging, games and more can be an HTML5 application*." (19)

We also see trend towards adoption of HTML5 among other, more stationary devices as well. With e.g. Windows 8, we see a trend towards native (or hybrid) HTML5 support among the PC operating systems. In Windows 8, developers can build HTML5, Metro Style apps. (20). Additionally, many members of the TV community (producers of smart TVs, Gaming consoles, Blu-ray players, set-top boxes and the like) have started to contribute to standardization efforts in relevant W3C groups, with focus on HTML5. (21) (22). This give reason to believe HTML5 not only will be the dominant among mobile apps in the future, but among all apps, across all devices. See Figure 3.

3.4 State of the art application technology

In this section we will present various examples of software solutions we regard as state-of-the-art, or SOTA, within each application category previously identified in section 2.3.1. We regard the following software as SOTA on the basis of them being of the most popular applications in the *cross-device* market. As with most digital products, providing the highest level of service is of upmost importance in order to gain a consumer's interest.

3.4.1 Category 1: Web browsing

Chrome Sync

Google Chrome is currently the most popular browser by far (23). Chrome is available on almost every device and platform as of today. Besides being a web-browser supporting many different technologies and APIs, its "Sync" feature is what makes it the state of the art software when it comes to web-browsing.

Functionality

The "Sync" feature of the Chrome web-browser allows the user to synchronize his/her Google services (currently open tabs, bookmarks, browsing history, extensions etc.) to the user's phone, tablet, and desktop programs so that it is accessible across any device (24).

(It should be noted that both Firefox and Opera offers similar functionality as well, with Firefox Sync (25) and Opera Link (26), respectively.)

Technology:

To make this sync infrastructure scale to millions of users, Google developers decided to leverage existing XMPP-based Google Talk servers to give them "push" semantics, rather than only depending on periodically polling for updates. "*This means when a change occurs on one Google Chrome client, a part of the infrastructure effectively sends a tiny XMPP message, like a chat message, to other actively connected clients telling them to sync*" (27).

3.4.2 Category 2: Multimedia

Netflix

Netflix is a popular, online, on-demand video streaming service, currently supporting over 400 different devices. Its software works on several devices such as smartphones, tablets, computers, streaming set-top boxes, blu-ray players, smart TVs, video game consoles etc. (28)

Functionality

With Netflix, a user can watch a movie or a TV show while logged into his Netflix account on the device he is currently using. If the user stops/pauses the stream or closes the application, he can resume the same video playback from the point he/she left off from the same or another device – as long as he is logged into his account.

Technology

Netflix claims that "*up to 70% of their code across all their cross-platform experiences is composed of shared code at every level of the stack in the form of infrastructure libraries and a UI framework*". We divide the Netflix technology into two main categories:

1. Architecture/UI
2. State management

Netflix is dynamically adapting the UI depending on the different types of devices it serves. In short, this is done using HTML5 and Qt/WebKit (their own browser rendering engine), as well as specific formatting and delivery engines (adapters – currently written in Groovy) on top of a generic Java API (29). See Figure 4.

The Netflix architecture "splits" the UI from the video player and low-level components. Netflix's technology of choice was HTML5. "*Because of it, Netflix can change their user interface easily anytime without redistributing a new client binary image or dealing with a review process*" (30).

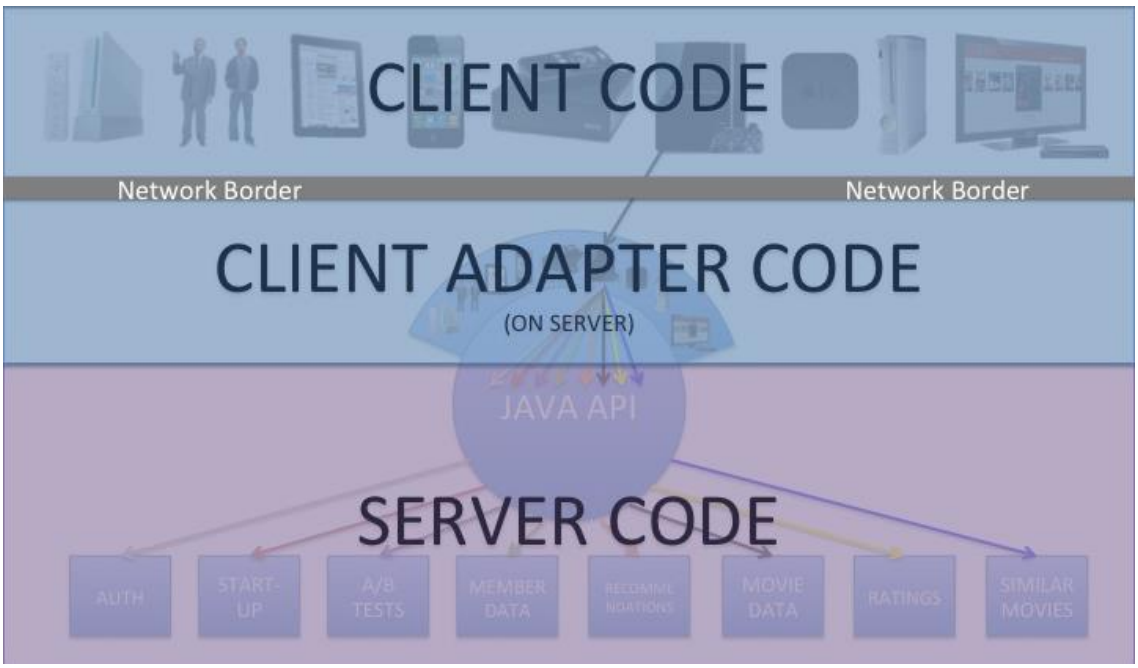

Figure 4. The Netflix Architecture. Taken from (29).

Netflix is based on a state management primitive called the "card". Each card manages the state of one or more views, which in turn contains components. The cards are firmly JavaScript-based, and can be stacked and traversed via the UI. This way, Netflix allows a user to retain his/her session across devices.

3.4.3 Category 3: Business

Google Docs (Drive)

Google Docs is a web-based office suite, provided by Google. It is housed by Google Drive, a file storage and synchronization service. It is cross-device, available on smartphones, tablets, and PCs (via the web browser).

Functionality

With Google Drive and Docs, a user is able to access his/her documents or files from any device he/she pleases. The user can edit documents in both offline and online mode, and has the option to take part in real-time collaboration with other users.

Technology

Google Docs uses Java on the server side, and is a web application with its own rendering engine, enhanced by HTML5 and JavaScript on the client side, while the files are kept at Google's Database. Unfortunately, Google doesn't reveal the details of their file storage system and the technologies they use. (31)

3.4.4 Category 4: Communication

There are a variety of very popular, cross-device communication applications, such as Skype and ooVoo. We chose Microsoft Lync, especially popular in the enterprise sector, due to the fact that it already provides session mobility, and as such is the closest related to our project's scope. Also, following Microsoft's acquirement of Skype in 2011, it is reported that Microsoft already has begun integrating Lync with Skype (32).

Microsoft Lync

Microsoft Lync (formerly known as "Office Communicator") is a voice, video, chat and document collaboration over IP software. It supports a wide range of devices, such as Windows PCs, Windows Phone, iOS, and Android smartphones.

Functionality

Lync enables a user to (33):

- Connect to the service with any device, from anywhere
- Have an automatically updated presence information (e.g. updated when you're away, in a call etc.)
- Communicate with one or several other users (friends, colleagues, partners etc.) in whatever way is best suited
    - IM
    - Video/voice chat
    - Audio call
    - Document collaboration
- Switch, forward or transfer calls to other devices without disruption to the conversation (34).

Technology

Lync implements the Session Initiation Protocol (SIP). SIP is used to establish multimedia sessions, as well as it provides session mobility (1). The actual telephone

and unified communication services provided by Lync is done via SIP trunking, a Voice over IP and streaming media service based on SIP (35). We will revisit SIP later in this chapter when discussing it in the related work section 4.2 (36).

Lync uses open media standards, such as H.264 Scalable Video Coding to provide "*a high-quality video experience on a wide range of devices*" (33). It is also believed Microsoft has plans of integrating Skype with Lync, as previously mentioned.

Lync Server requires a substantial infrastructure, consisting of many components, such as the mediation server, the edge server and the media gateway, which we will not cover in this paper. However, we can mention that Microsoft offers to deploy this in their cloud (Office 365), making it possible to connect to the service via the Lync Client from anywhere, without acquiring the infrastructure yourself (37).

3.4.5 Category 5: Games

Although Sony and Microsoft may offer the true state-of-the-art video gaming technology and experience with their coming video game consoles, PlayStation (PS) 4 and Xbox One, we choose to focus on another aspect of video gaming, namely cloud gaming. Though, it should be noted that Microsoft also has cloud gaming plans (38).

While it is common to see the same video games being available on PS, Xbox and computers, you will have to buy three different copies to play it on all these devices. Additionally, you can't resume your computer-saved game from the Xbox or PS, or vice versa. With cloud gaming, however, the idea is to provide the same video game experience to the subscriber, but giving the user the ability to choose between his/her devices.

Since 2010 there has been many new entrants in the fairly new market of cloud gaming. Wikipedia keeps a fairly good overview of these actors (39). One of the most used cloud gaming services, focusing on cross-device capabilities, is CiiNOW. (Another exciting cloud gaming service is NVidia's GRID (40)).

CiiNOW

CiiNOW (41), a cloud gaming platform, provides a cloud gaming service for Windows, Mac OSX, Linux, Android and a set-top-box. According to CiiNOW, they actually provide a 17% faster online gaming experience compared to local consoles.

Functionality

CiiNOW enables a user to play a variety of popular video games on whatever device the user wants, e.g. a smartphone, tablet, PC/Mac or TV Set Top Box.

However, we don't know of any feature allowing a user to automatically and immediately transfer his/her video game session between his/her devices with CiiNOW (or any other cloud gaming provider for that matter). Thus, the user has to manually launch the video game (or cloud gaming client) on each client in order to switch device.

Technology

CiiNOW, as well as other cloud gaming providers, utilizes Cloud Computing to provide "gaming on demand". The user accesses and interacts with the video game through a thin client installed on the device. This thin client captures the user's input and sends it to CiiNOW's hosting platform, or "cloud", where the actual game is stored, the user commands are executed, and the graphic is rendered. A video representing the state of the game is then created, adapted and streamed back to the device (42) . See Figure 5.

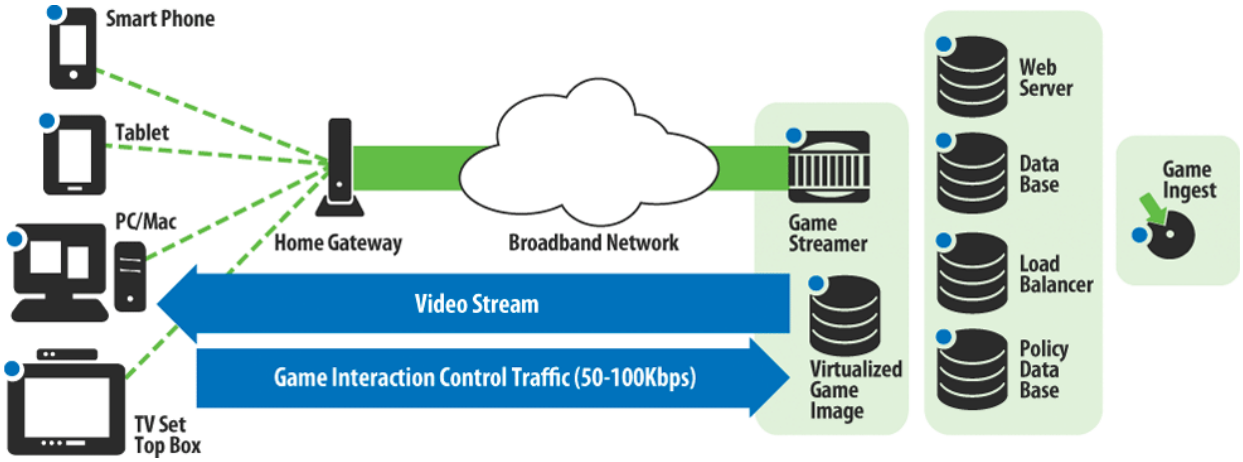

Figure 5. CiiNOW's Cloud Gaming solution. Taken from (42).

3.5 Conclusion

We've identified and seen that Cloud Computing and HTML5 are frequently utilized among the most popular cross-device and state-of-the-art application solutions.

Knowing what type of applications a cross-device session mobility platform should work with, we will move on to present both existing solutions and related research/work in the field of session/application mobility and see how they apply.

4 Related work

4.1 Introduction

In this chapter we will present both existing solutions and research within the field of session/application mobility. There is a lot of research focusing only on one application type/category (e.g. multimedia), possibly due to the different nature of applications. Fortunately, there are also some work proposing more generic solutions, i.e. a solution that should work with multiple application categories.

Common for all the proposed solutions we've encountered is a concept of session transfer, or session *migration*. An application session migration happens between the *source* device, i.e. the device currently being used, and the *target* device, i.e. the device the user wants to switch do. It entails some sort of transfer mechanism of the application's session/state on the source device to the target device, allowing the corresponding target device application to resume the session from where it was left off on the source device. Exactly how this migration is supposed to take place varies from solution to solution. Of the proposed solutions we have encountered, we can generally divide between two types of approaches:

1. Live migration
2. Cold migration

Live and cold migration are terms we have borrowed from the virtual machine community (43).

In the context of application session mobility, a *live migration's* goal is to retain the application session (with the other endpoint) throughout the migration, whereas a *cold migration* stops the execution of the application on the source device, and makes a new request (to the other endpoint) to start a new session from the target device. We will see that the chosen migration approach also influences the solution architecture to some extent, e.g. what entities are involved and where they reside.

According to (44), an architecture for a session mobility can be either network centric or device centric. In a *network-centric* approach, the network is responsible for taking care of the session migration from one device to another. "*The role of the user and the device is restricted to just providing some preferences and maybe advertise themselves."* (2). In a *device-centric* approach, however, the mechanisms and functions for realizing session mobility is put on the device, "*using the network as more or less an unintelligent infrastructure for data transfer*". While a network-centric approach requires high deployment cost as the complexity is put in the network, a device-centric approach requires more complex and capable devices and applications, but with a vastly reduced network dependency. This results in a trade-off situation. Where do we want the complexity to reside?

In the following two sub sections we will present and evaluate several solutions using a live migration and a cold migration approach, respectively.

4.2 Live migration solutions

In this section we will look at a couple of proposed solutions focusing on retaining the application session throughout the migration. Here, the goal is to avoid interruption of the original session with the other endpoint (e.g. application server), and to somehow transparently transfer this session from the source device to the target device. We will cover three types of approaches; first examine the implications of implementing support for SIP, then an adapter-based middleware-approach, and finally a proxy-based approach.

Session Initiation Protocol

SIP already supports session mobility, and has the capability of reconciling capability differences between the devices, i.e. codec, display resolution and bandwidth differences. It is an application-layer control (signaling) protocol for creating, modifying, and terminating multimedia sessions with one or more participants. These sessions include Internet telephone calls, multimedia distribution, and multimedia conferences. SIP runs on top of several different transport protocols. In total, SIP as an underlying application-level protocol can offer terminal, personal and service mobility. RFC 5631 (45) describes SIP's session mobility capabilities, while RFC 5589 (46) describes SIP's call transfer capabilities.

Using SIP, however, either requires both communicating endpoints to support SIP, or a SIP proxy would have to be used to interact with non-SIP application servers. Additionally, SIP only offers session mobility in regards to a media session, i.e. there are no additional state parameters sent during the session handoff, and thus SIP by itself cannot be used for stateful transitions for all types of applications (e.g. web sessions where among others, the history and cookies are important for a complete, stateful transition).

As many application servers are currently not equipped with the required SIP infrastructure to support SIP, some papers investigated solving the problem of session mobility using a SIP proxy approach (i.e. where the client is SIP-enabled, but the application server is not). The paper *Converged multimedia services in emerging Web 2.0 session mobility scenarios* (47) presents a hybrid-based architectural framework that uses a SIP integrated web client and a Converged (SIP and HTTP) Application Server. The paper presents technical contributions of a SIP-based hybrid architecture that leverages SIP, HTTP and XML to provide converged services. The solution is a

Mozilla Firefox-extension, named TransferHTTP, implemented as a SIP-stack. The SIP stack is implemented in both the client and the proxy. Upon session transfer, "*the session data of a Web session transfer request is sent in an XML format using the SIP MESSAGE method, and it could consist of a URL, cookies and session tokens depending on the kind of request.*"

Thus, by implementing an extension utilizing the SIP MESSAGE, SIP can be used to provide session mobility for applications that require additional state information. Such a solution requires a SIP-based deployment environment, i.e. the clients need to be SIP-enabled.

Evaluation:

We argue that the value of SIP lapses if you only can use it within a proxy domain. Additionally, SIP alone does not support complete session mobility for all types of applications (only media sessions), thus one would have to implement an extension to provide this, along with defining a device discovery protocol, as that is outside the scope of SIP. Another drawback of session mobility through SIP is that the signaling is relatively heavy and requires text messages of considerable size. Thus, we argue it is not the ideal basis for a generic session mobility solution.

Middleware-approach

A Flexible Framework for Complete Session Mobility and Its Implementation (48) provides session migration while retaining the original session with the other endpoint. Their solution is middleware-based, where each device has a central component, the Relocation Manager (RM), which coordinates the application session transfer between two devices, using three adapter components (see Figure 6):

1. The *network adapter* triggers an update of the IP address to redirect traffic, utilizing e.g. Mobile IP, Host Initiation Protocol and/or a shim.
2. The *transport layer adapter* extracts all transport layer specific info from the OS. In case of the TCP transport adapter all necessary state information is extracted from the OS in order to re-establish the socket on the other device.
3. The *application adapter* takes care of state extraction from the applications. This requires the application to provide an application adapter specific interface for state export and import.

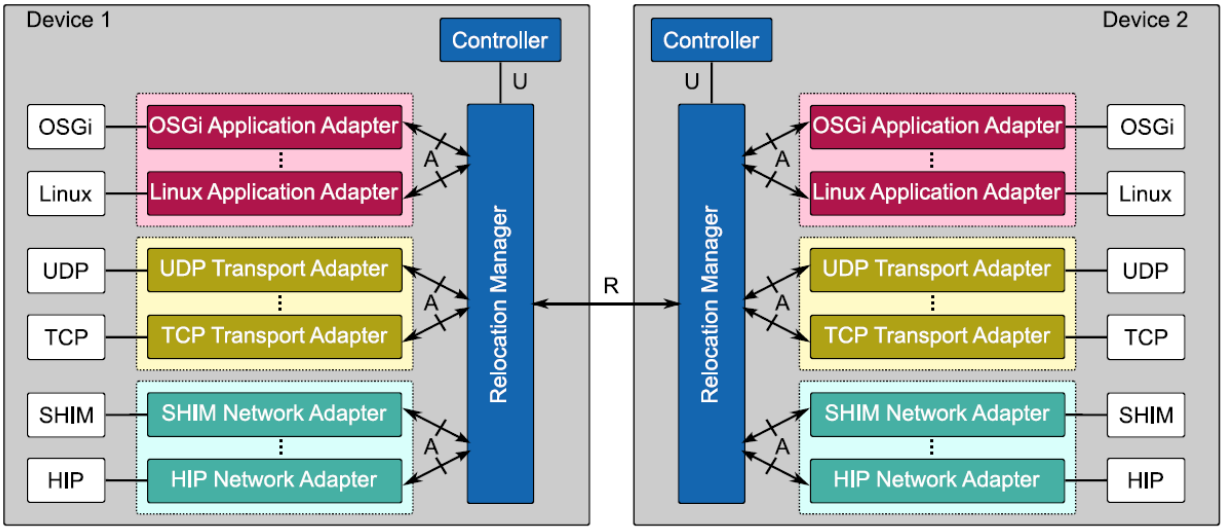

Figure 6. Architecture overview, showing the Relocation Manager and the adapter components.Taken from (48).

Required of the applications that were to utilize the migration functionalities provided by the RM, was to:

- implement an interface to the application layer adapter, and subsequently a UI enabling the user to trigger migration functions.
- support session extract and import methods, which, depending on the application, could mean application modification on both client and server-side.

Evaluation:

Using low-level operating middleware, the original application session is transferred, providing a seamless and transparent migration. However, the authors identified the following unresolved issues:

- Since switching of IP addresses in active TCP connections is uncommon in today's IP networks, entities in the network that track the connection state (firewalls, NATs) cannot associate address switched TCP packets with connections and will block traffic
- They considered transfer of TCP state with Linux only, while transferring TCP state between operating systems with different stack implementation demands for suitable transformation of state information in order to work correctly.
- Uncertainty regarding performance and security issues

Additionally, the prototype was tested and verified in a homogeneous environment (Linux-Linux) only. If the solution is to support cross-device mobility we have identified the following points which the authors fail to discuss:

- How content adaptation will be handled. As the target device may have different capabilities than the source device, we see the need for a potential adaptation of the application data content sent from/to the other endpoint (e.g. the application server).
- Who should be responsible for this content adaptation? Is it feasible, or even possible, to have the adapter components on the device do this, or do we need to introduce an external entity?

Thus, their solution has potential, but certain issues need to investigated and solved before it can be regarded as applicable to use in a cross-device session mobility solution.

Proxy-based approach

In *Proxy-based Hand-off of Web Sessions for User Mobility* (49), they present a protocol built atop HTTP to provide session mobility for web sessions, exploiting a proxy-based architecture. Here, a proxy entity is placed between the clients and the application servers. The authors of the paper chose the proxy approach to have *"a minimal invasiveness on the legacy distributed application*". When a user is authenticated by the proxy, a unique dummy agent is generated for him, which is used to represent the user and his devices to the application server. The servers see the client as that unique dummy agent representing the device, even when the client migrates from a device to another.

A similar, but more generic approach is proposed in *Session mobility solution for client-based application migration scenarios* (3). It presents a Migratory Service Platform (MSP), a middleware platform *for migratory applications* based on the OPEN project (50) (51). The MSP contains a server-side where all the common migration functions are located (keeping them centralized), and a lightweight client-side running on the end-users device. The client-side is kept lightweight by interacting with the platform and thereby does not have to contain migration functions themselves. The *migratory applications* interact with the MSP through a specifically defined interface. The Migration Server (MS) may reside anywhere (e.g. on any device or accessible via the Internet) as long as it's reachable by all clients. See Figure 7.

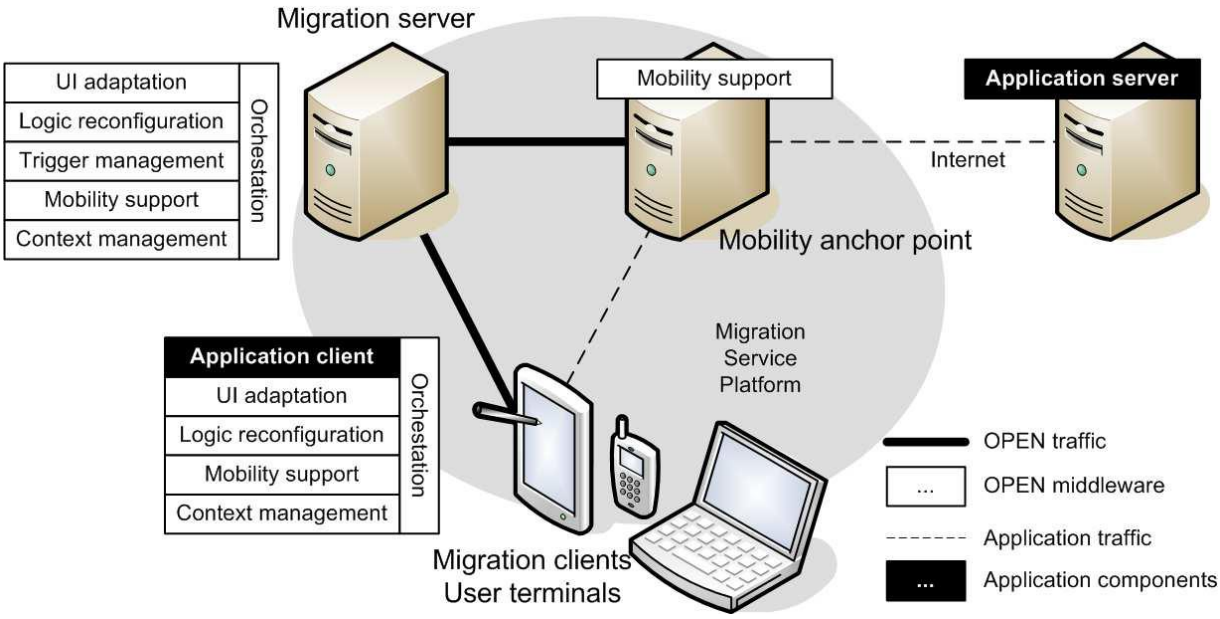

Figure 7. The Migratory Service Platform. Taken from (3).

In the MSP architecture they introduce a SOCKSv5-based proxy (52), which serves as a Mobility Anchor Point (MAP) to provide mobility support. It is controlled by the MS and handles seamless handover of connections during a migration, which consequently are kept transparent to the remote application servers.

The paper argues the MSP provides all the three major aspects within application migration; context change ("*where context is any information that can be used to characterize the situation of an entity*"), adaptation (based on context information) and continuity. To achieve those goals, the MSP of the OPEN framework is based on a series of components that focus on the following aspects independently:

1. User interface adaptation
2. Application logic reconfiguration
3. Mobility support (Network components)
4. Context management
5. Migration orchestration
6. Trigger management.

An application migration is then done either manually by the user, or as a reaction to contextual changes. MSP supports both full and partial migration. A partial migration could e.g. be the migration of only the user interface to another device. When migrating, the application is paused on the source device, the state is retrieved by the MSP, then the application is initialized at the target device, where it receives the *adapted* state by the MSP and the application is resumed, and the session is continued. Additional mobility support (redirection of network traffic) is also done at the MSP. See Figure 8.

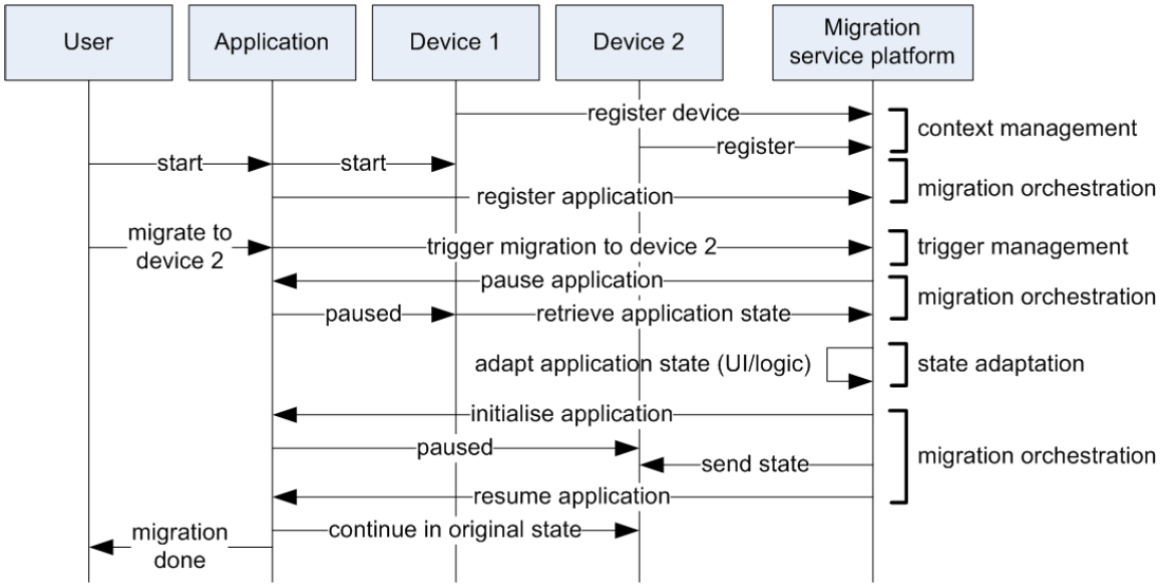

Figure 8. Steps in basic migration procedure provided by the MSP. Taken from (3).

Evaluation:

The proposed proxy-based solution adds components that take care of necessary adaptation to make migration possibly despite network and device constraints. However, the migratory application has to fulfil a number of requirements to work with OPEN, i.e. it needs to implement the OPEN application interface to interact with the MS, as well as implement network connections in order to interface to the MAP. An obvious drawback of using SOCKSv5 is that it only works for TCP and UDP transport protocols.

However, we consider the proposed solution design interesting for our use. But we argue one should especially investigate the feasibility of the UI adaptation, logic reconfiguration and context management modules. Is it feasible to provide proper adaptation and context management in a generic context, i.e. for all types of applications? Additionally, the authors don't really discuss how they handle the migration with respect to the application servers. Since the migration is kept transparent to the application servers, the session is never stopped. What happens to the application data sent from the application servers during and after a migration, in the case of switching to a new device with new capabilities?

We also identify the need for investigating the security aspects of having a proxy act on the behalf of the user agent. If the solution is to work with applications working with sensitive information, we need to ensure the platform's, and especially the proxy's, security and privacy levels.

4.3 Cold migration solutions

In this section we will look at a couple of proposed solution where the application session is *not* retained throughout the migration, i.e. where the original application session with the other endpoint (e.g. application server) is terminated on the source device and started again from the target device (cold migration). Here, the goal is to enable the application on the target device to launch in a "stateful" manner, i.e. to make the application start from the point it was terminated on the source device. An approach like this is used in (2), where a video conference session is allowed to be migrated between (heterogeneous) devices. However, that solution is specifically designed for a video conference service, and thus fails to be generic.

We will first take a look at an existing solution, DIAL, with interesting, relatively generic session mobility functionality. Though, designed as a 2nd-screen protocol, DIAL has its limitations. We will then present another proposed solution, which, unlike DIAL, is designed for application mobility.

DIAL

DIAL (53) (Discovery And Launch), developed by Google and Netflix, is a 2nd screen protocol for discovery and launch of applications. DIAL "*enables 2nd screen applications to discover and launch 1st screen applications on 1st screen devices*". It is based on SSDP (Simple Service Discovery protocol) as defined in UPnP and HTTP. DIAL is an interesting solution, as it already has secured support from key consumer electronics makers, content services and app makers, according to (54).

Service/functionality example:

Use your Android* smartphone or tablet to search for a video on www.youtube.com . Play the video on either your handheld device or migrate it to your Google TV* and use your Android device as a remote. Note that the latter remote-like functionality is application-specific and outside the scope of DIAL.

*Available soon on additional devices.

In DIAL, the migration of an application works by having the source (2nd-screen) device send an argument string describing the application's state, to be passed to the target (1st-screen) device application on launch (see Figure 9). The format of the argument, and how the argument is passed, is application specific, and thus of the scope of DIAL.

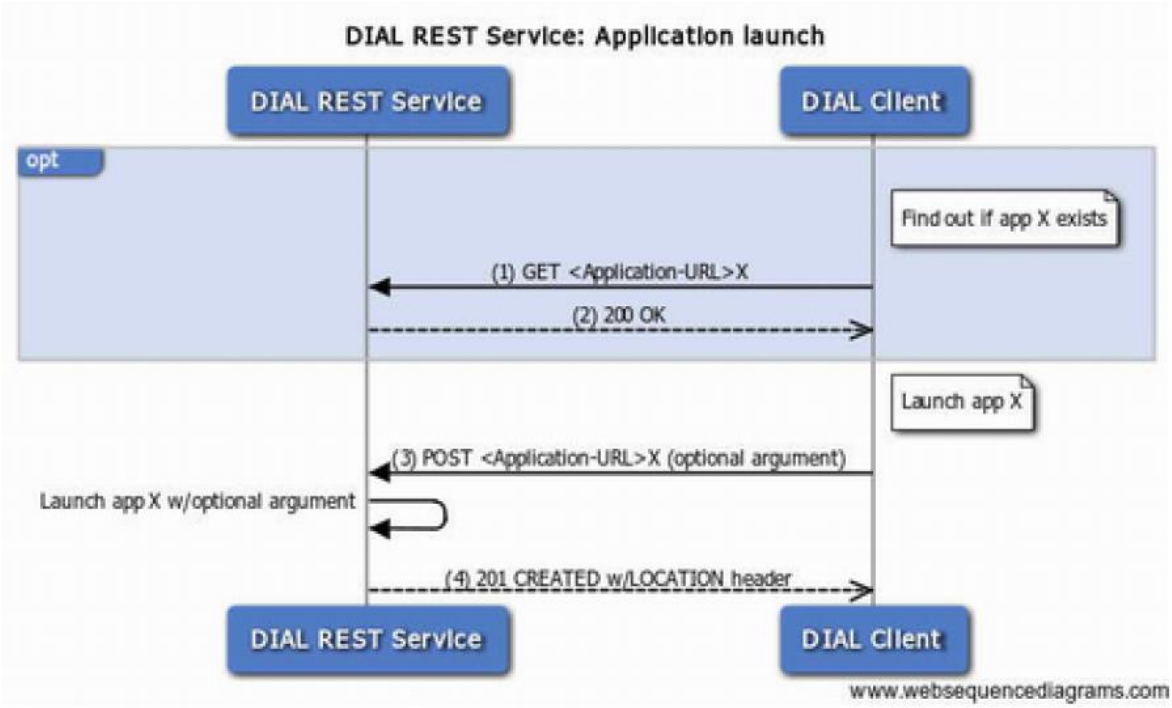

Figure 9. DIAL application launch. Taken from (53).

Evaluation:
They way DIAL handles the migration, by having the application developer implement and define functionality for retrieving, describing and importing the state, is very interesting. This way, the solution becomes very generic, as it is up to the application developer to implement it as they see fit, while DIAL only provides the device discovery and communication parts. Using DIAL will require application modification, but after seeing how many actors were willing to conform to DIAL's standard in order to use its functionality, we believe this is a reasonable requirement.

However, mainly due to the fact that it is designed as a 2nd-screen protocol, DIAL has its limitations when it comes to being a complete session-mobility solution:

- It works only between devices in the same LAN environment, i.e. not cross-domain.
- The migration is one-way, i.e. it can only occur from a "small screen" (2nd-screen) to a "big screen" (1st-screen) device, but not the other way around. This means the devices an application can migrate to and from are pre-defined.

A2M

In *Context-aware Application Mobility Support in Pervasive Computing Environments* (55) they present a novel architecture, Application Mobility Manager (A2M), to provide seamless, cross-device and cross-domain application mobility. Using A2M, they let an

application follow a user while he/she roam between several networks and devices. The A2M is divided into three components (see Figure 10):

1. The *Migration Manager* (MM). The MM is the system's foundation, and provides a generic interface to the application running on top of it. It is installed on each device and is responsible for device advertisement and discovery, for context collection and for carrying out the actual migration.
2. The *Application Adapter* (AA). The AA encapsulates the application and provides it with semantics such as state information. It is responsible for adapting the application based on the device's specifications/capabilities.
3. The *GUI Adapter* (GA). The GA is responsible for modifying the application's GUI to best fit the device currently in use.

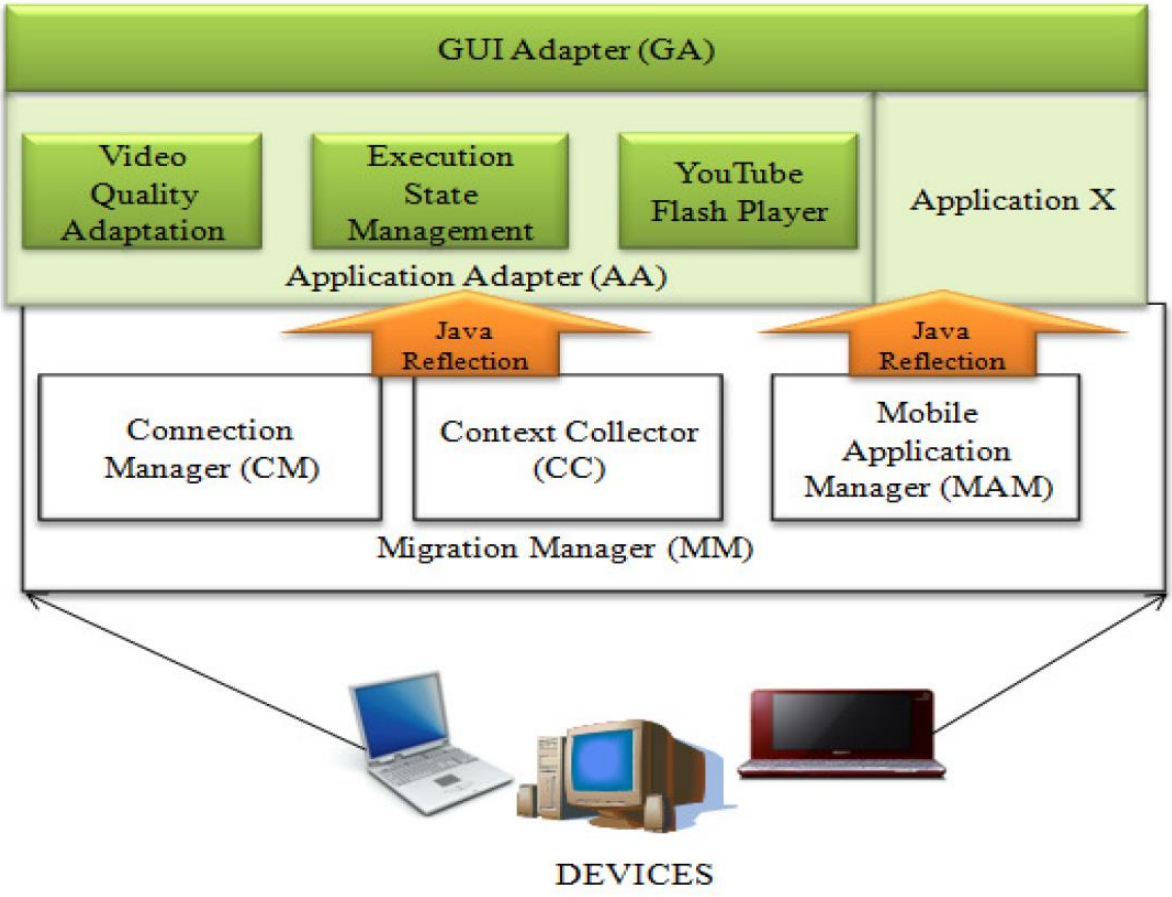

Figure 10. The Application Mobility Manager (A2M). Taken from (55).

In A2M, the devices are connected to each other in a peer-to-peer manner, where at all times one of the devices acts as the "host", or the server component. This allows for a decentralized architecture, where at the same time devices can advertise themselves, as well as discover each other.
Upon a migration, the source and target device establish a TCP socket connection over which the application is migrated. The migration procedure proposed in the paper is based on the concept of (Java) Reflection. Reflection enables you to examine and/or modify an object at runtime (56). Here, Reflection is used to retrieve a file (or files)

describing the application's current state/runtime, which subsequently is transferred to the target device over the socket connection. Consequently, by Java Reflection, "*the newly downloaded application can be started on the new device from where it was suspended on the previous device"*.

Evaluation:

We identify several issues with the A2M when considering it in the context of a generic, cross-device session mobility solution, especially regarding the Reflection-based migration approach and the layered design. First of all, the Reflection-based session migration approach puts constraints on the applications the solution can work with, both on their development environments and their design. I.e. they require the applications to be developed in a programming language supporting reflection, which not all programming languages do. Secondly, due to the Application Adapter and GUI Adapter components, they place significant constraints on the way the applications are to be developed. We argue these constraints conflict with a generic solution, and should be avoided.

However, we consider the Migration Manager component, and especially its peer-to-peer and socket connection modules interesting. With this, they are able to "*provide users with seamless video Experience*" with their Mobile YouTube Player demo application, using a scheme which is effectively stopping the application on the start device, transferring it over this socket connection, and starting it again on the target device, resuming at the exact same point.

4.4 Conclusion

We've differentiated between two overall approaches when it comes to a session mobility solution - live and cold migration - and we've seen that they affect the architecture of the solution to a degree. In a live migration the original application session is retained throughout the migration. While this may be the most generic approach, we've identified potential issues in regards to content adaptation and security. In a cold migration, the original session is terminated on the source device, and started again on the target device. While this can be an effective, lightweight scheme, we need to investigate its application applicability when it comes to being generic, given certain implications that an application type can pose on us, such as real-time communication applications.

When discussing our solution design, we will revisit and take inspiration from many of the concepts introduced in this chapter. However, before we start the design discussion, we want to identify the wants and needs of those who actually are going to use the potential solution. What functionality is important to the user?

5 User Study, Use Cases and Requirements

5.1 Introduction

Before designing a solution, we should ensure ourselves that the solution we are about to design (and later implement) is something the end users deem valuable, i.e. a solution providing functionality the end users both *want* and *need*. E.g. what application categories and types does the end user think benefit from session mobility?

That's why we decided to carry out a user study, both by holding an office discussion with some fellow students, as well as performing a user survey on a carefully selected group of people. From the user's input we've derived use cases, and consequently requirements that together create the backbone for our solution design.

5.2 User study

Before creating the survey, we sat down and talked with ten other students from our university – all of which come from a technological study background. We did this to find out what *they* believed a cross-device session mobility platform entailed, as well to help us get an idea of how end users may perceive the idea of providing this to applications. Based on the conclusions from the office discussion, we created a user survey and performed it on selected users from different backgrounds.

5.2.2 Office Discussion

The office discussion was very open, and everyone contributed with their own opinions and ideas. During the discussion we covered most aspects of a cross-device session mobility platform. Summarizing, the following are the major conclusions from the office discussion.

General applicability notes

- Its use was deemed most, if not only, applicable for local areas, like at home, or possibly, at work, where you have multiple devices in the vicinity. In other situations, e.g. when commuting, or when you only have one device, a manual resume was preferred.
- Although closely located, some devices, typically mobile, are not necessarily connected to the LAN at all times. As such, the need for a cross-domain capability was identified. Having the service being cross-domain was seen as especially useful in cases where a user either enters or leaves a "home environment", and wants to transfer his currently running application session from or to his mobile device, respectively.

Application category consideration

- Cross-device session mobility was thought to primarily be beneficial for *commercial entertainment services*, especially *multimedia streaming*.
- Business and communication applications can also benefit from session mobility.
- Browsing was deemed good as it is now, with Chrome's Sync feature mentioned as a good example.
- Cross-device videogame mobility was unwanted in general, due to suspected problems with resolution, control mechanisms and performance.

Wanted features:

- An "External view" feature, i.e. a partial migration feature, was wanted for document presentation/viewing, multimedia streaming, and photo album viewing. This entails remotely controlling the view of an application, e.g. controlling a video streaming application displayed on the TV via the smartphone.
- A "full migration" feature was wanted for multimedia streaming and voice/video calls, i.e. the ability to completely transfer, or migrate, an application session from one device to another.
- Ease of use should be emphasized, i.e. it should be easy and intuitive for a user to start either an "external view" or a "full migration" session. As such, it was argued that the controls for doing this should be available via the applications UI.
- The application on the target device should be fired up automatically upon an incoming migration request. It should not be required that the application is already running on the device.

5.2.3 User Survey

To gather input from people with different backgrounds, as well as ensuring we would get replies, we personally contacted and requested a selection of friends and family to answer the survey. We chose people of different ages, technical background, education etc., in order to hopefully emulate a normal user demography.

For carrying out the user survey, we used an online questionnaire. In the hope of gathering as many answers as possible, we tried to keep the survey short and simple, giving the users the choice to reply briefly, or, if wanted, to provide longer answers. First, we shortly introduced the reader to the concept of cross-device session mobility. The actual survey consisted of 7 questions, three of which we simply used to gather statistics of the replying users. The last four questions was divided into two parts.

The first part consisted of two grading questions related to the identified application categories, as well as functionalities and features related to a type of migration. Here, we introduced to the reader two types of migrations:

1. *Full migration:* The process of moving an application to a new device while terminating it at the device it was originally running.
2. *External View*: The process of moving an application to a new device, while controlling it from the device it was originally running.

In the second part we asked for the user's input - how did they imagine/want an application transfer to take place? Could they think of any other situations where session mobility could be useful?

For more information about the survey tool we used, or if you want to take a look at the survey, we refer to appendix, section 12.1.

Results

Rather surprisingly, document editing and viewing applications scored highest on usability, followed by browsing. However, the rest of the application categories scored almost as high, except from online video games. Thus, we will not focus on video game applications when developing our use cases and requirements to the solution.

Of the functionality grading, both the "full migration" and the "external view" features scored high, with "external view" scoring highest. This shows that such a feature is especially attractive, and thus should be given a high priority in a complete solution.

As expected, the replies when prompted to answer with text was usually very brief, but at the same time comprehensive. When asked about how they wanted an application transfer to take place, the two most frequent words amongst the answers were "automatic" and "wireless". We chose to interpret this as a general need for a simple, automatic migration functionality, e.g. by having most of the migration procedure to execute automatically upon a user trigger.

5.3 Use cases

In this section, we present two scenarios, containing several use cases, describing how our solution should work. These use cases are derived and based on the needs and wants of the users, as well as the other research we have done. The use cases will in turn help us to identify both the functional and non-functional requirements of our solution.

Scenario 1: Immediate Transfer

Actors:	User
Prerequisites:	The user running an application on a source device (e.g. desktop computer, laptop, mobile, tablet)
	Both source and target devices are turned on and have internet connectivity.
Description:	1. The user decides to use the "Migrate" command and gets a list of all the active devices registered to the same user who are able to run the application he is currently running on his device.
	1.1 The user decides to use the "Immediate External View" option on a selected target device to transfer and launch the current active application session to this target device, while maintaining the control of the application at the source device. The application is launched on the target device, and resumes the application session immediately. The user can now control it with input from his source device.
	1.1.1 The user sends an application-specific command from the source device to the target device, which is executed on the target device application
	1.1.2 The user decides to "Exit" the "External View" session, which transfers back the current view from the target device and resumes the application on the source device. The application on the target device is closed.
	1.1.3 The user decides to "Finish" the "External View" session to finish/complete the migration to the target device. The application on the source device is closed, and the user can use normal input methods on the target device to control the application.
	1.2 The user decides to use the "Immediate Full migration" option on a selected target device to transfer and launch the current active application session to this target device. The application is launched on the target device, and resumes the application session immediately. The application on the source device is closed, thus the user can only control the application from the target device.
Applicable for:	Video/Voice chat, Web-browsing, Document Viewing and Multimedia applications when the target device is in the same room.

Table 1. Scenario 1 use cases

Scenario 2: Suspended Transfer

Actors:	User
Prerequisites:	The user running an application on a source device (e.g. desktop computer, laptop, mobile, tablet)
	Both source and target devices are turned on and have internet connectivity.
Description:	2 Same as in scenario 1, step 1
	2.1 The user decides to use the "Paused External View" option on a selected target device to transfer and launch the current active application session to this target device. The application is launched on the target device, but resumes in a paused mode. The user can now control it with input from his source device.
	2.1.1 Same as in scenario 1, step 1.1.1
	2.1.2 Same as in scenario 1, step 1.1.2
	2.1.3 Same as in scenario 1, step 1.1.3
	2.2 The user decides to use the "Paused Full migration" option on a selected target device to transfer and launch the current active application session to this target device. The application is launched on the target device, and resumes the application session in a paused mode. The application on the source device is closed, thus the user can only control the application from the target device.
Applicable for:	Non-real-time applications, where the target device is not necessarily in relative proximity.

Table 2. Scenario 2 use cases

5.4 Requirements

Based on the use cases, as well as the existing solutions and related work we've visited, we have derived a set of requirements for our solution. Since we want a generic solution, we use common ground between applications belonging to the same application category when we phrase the requirements for a certain application category.

Table 3 lists the identified requirements for a generic, cross-device, cross-domain session mobility platform.

ID	Short name	Description	Priority
Mob1	Session Mobility	The solution must provide a transfer mechanism enabling the user to transfer his active application's session between devices upon request.	H
Mob1.1	ImmPaused	The solution must be able to perform a migration in either a paused or an immediate mode, upon user request.	H
Mob1.2	ExtView	The solution must be able to establish an external view session, where the target device application is controlled from the source device application	H
Mob2	Cross-device mobility	The solution must be able to work cross-device, or cross-platform, i.e. it must be able to work on multiple computing platforms (mobile, desktop, stationary etc.), provided the application can run on the devices.	H
Mob2.1	Cross-device Mobility21	The solution needs to be able to identify the available devices that have installed the application currently running on the source device.	H
Mob3	Auto-launch	The solution must be able to launch the requested application even if it's not currently running on the target device.	H
Mob4	Cross-domain mobility	The solution must be able to work cross domain, i.e. devices associated with different access points	H

		and/or security domains must still be able to see and migrate to each other	
Mob5	Real-time Mobility	The user should be able to use real time applications and change terminal/device anytime, without violating the real-time application's time constraints.	H
Hist1	Cross-device History 1	The user should be able to trace back his past actions in the application after changing device, if applicable.	H
Dis1	Device discovery	The solution must provide a way for the device to advertise itself as well as discover other devices registered to the same user.	H
Int	Application Interface	The solution has to provide the applications with an interface the application developers have to implement in order to access the various migration functions. This entails enabling the users to trigger migration functions directly from the application's (preferably intuitive) UI.	H
Wb1	Web-browsing 1	The relevant files (e.g. cookies, browsing history and bookmarks) from the source device's application must be transferred to the target device.	H
Wb2	Web-browsing 2	The service needs to be aware of how many active sessions the user has (logged in, submitted forms, tabs, scrolling loc.)	M
Mul1	Multimedia 1	A message with information about the currently playing media file's ID and its current position/time must be sent to the new device. If the application on the source device has a play queue, this should also be transferred to the target device.	H
Mul2	Multimedia 2	The multimedia playback must be paused/stopped during the process of migration to ensure it starts on the correct time at the target device, after the migration is completed.	M

Doc1	Document 1	A document's revision history must be available to both source and target devices. If the document and document revision history lie locally in a device, the solution will take care of the file transfer to the new device.	M
Com1	Communication 1	The transfer of a communication session needs to be handled in a way that maintains the connection between the two parties and doesn't affect the other endpoint's experience during the migration/call transfer.	H
Com2	Communication 2	The message History of an IM application needs to be available on both source and target devices after a transfer.	L
Com3	Communication 3	The session handover of an ongoing call has to occur without significant delay.	H

Table 3. Solution requirements

5.5 Conclusion

Based on our findings from the user's input, we have concluded that a cross-device session mobility platform certainly provides functionalities the users will value. It is perhaps of must use in the home environment of a user, where he/she has several devices available in the near vicinity. As such, it would be natural to think it will primarily be used for entertainment purposes, i.e. multimedia applications, though all application categories (except for the gaming category) was thought to benefit from session mobility in one way or another.

The users valued both the "full migration" and the "external view" features, finding the latter especially attractive.

From the results from the user input, we derived use cases, and consequently requirements for our solution, where we've tried to prioritize the different functionalities and features following the user's wants and needs. Our solution design will be based on these requirements, where we will aim to satisfy each requirement.

6 Design discussion

6.1 Introduction

We're after a generic, cross-device, cross-domain session mobility platform solution. The solution should be a mechanism that allows you to select among other available devices on which you want to continue your ongoing application session. The actual migration should happen seamlessly, and the application should start automatically on the target device, continuing the session. The latter would require some sort of background service running on the device, capable of receiving the session transfer request, and consequently launch the requested application.

One of the overall goals of the project is to make it as generic as possible. I.e., not only should it work with different types of applications, but it should preferably work with various types of applications *within* each category as well. E.g. two video streaming applications may be implemented in very different ways. We want our solution to work with both of these.

Another goal is to create a solution that requires minimal modification of the actual application. Though, in order to be user friendly, we argue the user should be able to trigger the various migration functions via the application's UI. This will require some modification of the existing application client, e.g. implementation of an interface to the session migration functions. Our hope is that this will open the possibility of our solution working with third party applications as well. While an existing third-party application would have to be modified, new third-party applications can be developed to work with our solution in the first place, and thus will require no actual modification per se.

In order to design and specify a solution, we will take inspiration from, and discuss possible adjustments to, both the existing and proposed solutions we presented in chapter 4. We will try to extend or combine the ideas to reach our goal of providing a generic, cross-device, cross-domain session mobility platform, satisfying the requirements specified in chapter 5. We will weigh their advantages and disadvantages, and finally make a well informed decision.

We divide the design discussion into two parts:

1. **Migration.** The solution must provide a way of realizing the actual migration, having the target device application continuing the session from where the source device application left off. This also includes having the target device automatically launching the requested application. We will present one live and one cold migration approach. Here we will assume there is an overall architecture supporting the migration, but we will not discuss it in detail, unless there are central components that need to be introduced.
2. **Architecture.** The solution must include a way for devices to connect/register to the solution, where they can advertise themselves, as well as discover other

devices. Additionally, upon a migration, the solution must provide a platform for communication/transfer between the target and source devices. We will discuss the alternative architectures and chose an architecture we believe is most suited to our chosen migration approach.

6.2 Migration

The migration phase entails the transfer of the application session from source device to target device. We will differentiate between the so-called live and cold migration approaches, as introduced in the related work sections 4.1 and 4.2. While these approaches vary a lot, they do have some common ground.

In the following subsections, we will first discuss these common parts, before we present various ways of realizing live and cold migrations, with the goal of providing a generic cross-device, cross-domain session mobility platform. We will discuss their applicability and feasibility, and if needed, investigate possible workarounds. Throughout this section we will assume there exists an overall architecture/platform supporting device discovery, session transfer etc. Finally, we will conclude this section with the migration approach of our choice. Then, we'll try to find the most suitable architecture to support such a migration.

6.2.1 Common ground

We have identified the following common ground between the live and cold migration approaches:

- *Session/state depiction*. There needs to be some sort of application session/state description to be transferred and passed to the target device application in order for it to prepare and/or continue/resume the session
- *Automatic launch*. The target device needs to be able to launch the requested application, even if it's not running, as well as pass to it the state/session depiction.

State/session depiction

Whether it's live or cold migration, the target device application needs to be informed about the session it is about to resume and/or the state it is about to enter. Thus, some sort of session/state depiction needs to be communicated to the target device application upon launch.

We've seen several ways of doing this, from low-level (network, transport and application layer) state export/import to Java Reflection to an application-specific String-representation (DIAL). Although the latter may impose application modification, we argue it is the most generic approach, as it becomes up to the application developers

to retrieve and depict the application session/state, as well as to launch upon this depiction. Hence, we will base our solution on this scheme.

To prove our point, consider developing a generic session/state template to which all applications working with our solution have to conform to. First off all, developing such a generic template could become a very complex task, considering the different nature of applications. Secondly, it will still require modification of the application, without the freedom of implementing it in the way that is best suited to the given application.

Unlike DIAL, which sets an argument string size limit of 4KB, as well as format restraints (e.g. the character encoding shall be UTF-8 (53)), we'd like to:

- Set no state/session depiction size limit (though we would urge the application developers to restrict the size as much as possible, as it directly affects the time it will take to migrate).
- Set no constraints on the format of the state/session object(s), or data structure(s), except requiring that it has to be serialized before it is transmitted over the network.

We argue this will allow for a high degree of freedom for the application developers, as they now can depict the session/state, as well as implement the retrieval and launch methods, in any format and way they see fit.

Automatic launch

One of the requirements for the solution is that the application on the target device is to be automatically launched upon a migration. This requires either the application to be already running on the target device, or some sort of background service to be running, able to launch the application upon request. We argue that in most cases the requested application will *not* be running on the target device, and that requiring it to do so will lead to the applicability of the solution to lose its meaning.

Thus, in order to launch a non-running application we need our solution to interface to the *native* features (hardware, OS, etc.) of the device, thus requiring the client-part of our solution to be native as well. While it may be cumbersome to implement a native clients for each device, we argue it is necessary, if we want out solution to provide the "automatic-launch" feature.

6.2.2 Live migration approach

A live migration keeps the original session alive throughout the migration. In chapter 4 we saw that this can be realized by either:

- Making the application "migration-aware" by adding session mobility support to the application, e.g. by adding SIP support on all entities involved
- Making the migration happen transparently to the other endpoint (e.g. application server or remote peer), with e.g. a middleware- or a proxy-based approach.

We don't find SIP applicable for our solution, as it would require substantial application modification by adding SIP support on both client- and server-side. This clearly contradicts with our goal of minimizing the amount of application modification. Additionally, it only supports session mobility for media sessions, and would need to be extended should it provide a more generic session mobility.

A possible workaround to a "migration-aware" approach is to make the migration procedure *transparent* to the non-migrating endpoint. This way, our solution does most of the work, and we minimize application modification.
In chapter 4, we presented a middleware-approach where adapters on the network, transport and application layers were utilized to seamlessly and transparently transfer an application session from one device to another. However, the solution had several issues, regarding NAT/firewall traversal, security and last, but not least, cross-device compatibility. If used to migrate cross-device, we identified the need for an entity that may have to perform content adaptation, but no such need nor entity was presented. We argue performing content adaptation on the device is not feasible. It is unlikely it supports all formats it may have to transcode between, but regardless, it is unwanted, as it can be very heavy, at least for a low-end devices. Thus, we would need an external entity performing the content adaptation. That way it makes sense to move the migration functions near the entity as well, moving the complexity away from the device. In chapter 4.2, we presented a proxy-based approach, based on the OPEN project. This will serve as our main inspiration when considering a live migration solution.

Proxy-based live migration approach

In a proxy-based approach, a proxy entity is introduced, placed between the devices and the other endpoint, with the following main responsibilities:

- It should act on behalf of the devices by generating a unique dummy agent for each user, and all his/her devices. Thus, the other endpoints see the client as that unique dummy agent representing the device, even when the client migrates from a device to another.
- It should orchestrate application migration between devices, transparently to the other endpoint.

The other endpoint may either be an application server (e.g. video streaming server) or a remote peer (e.g. a videochat peer). As the migrations are performed within the "proxy domain", they will happen transparent to the other endpoint.

Let's consider what happens during a migration. Upon migration, the user, from the source device's application UI, selects a target device to which he/she wants to continue the application session. Since the proxy is to orchestrate this migration, it is naturally to have the signaling go through the proxy as well. The target device will receive the migration request, as well as an application-specific session/state depiction, enabling the target device application to continue/resume the ongoing session. When the application is ready to resume the session, the proxy gets notified. Now, application data sent from the other endpoint gets forwarded to the target device instead of the source device, and the application session is continued without the other endpoint knowing about the migration event. In chapter 4, we identified two issues with such a scheme:

1. Application session handling during a migration
2. The potential need for content adaptation

Application session handling during a migration
First of all, it is important to retain the session during the migration, i.e. the proxy may have to send signaling/control data to the other endpoint throughout the migration. This is especially true for RTC-apps, as we need to take into account the user experience of the remote user. This can be handled by e.g. not stopping the application on the source device before the target device application is ready to resume, which should ensure a natural flow of messages throughout the migration. But what happens with application data content sent from the other endpoint? We can consider two alternative ways of minimizing application loss during a migration:

1. Having the proxy *cache* incoming data during the migration, if applicable. This requires application profiling, so that the proxy knows which application data to cache and not (e.g. there are no use in caching real-time comm. data, and in this case some RTC data will be lost).
2. Having the source device app. send a pause/hold request before migrating (if applicable), pausing/holding the session until the target device app. is ready and will send a play/resume to resume the session. This will ensure lossless migration, as well as being a less complex and proxy-demanding solution when compared to caching. It may require some applications to implement these pause/play features if they don't already have them, but we argue these are common features in both streaming and communication applications, and as such modification will not generally be the case.

Hence, we should be able to retain the application session fairly well during the migration procedure, by pausing/holding the application session upon trigger, and by not stopping the source device application before the target device is ready to resume.

Content adaptation
The second thing we need to investigate is the potential need for content adaptation. After successful migration, there is now an application session between the target device and the other endpoint. As we want a cross-device solution, this target device's

capabilities and properties may be very different from those of the source device, when it comes to e.g.:

- Device and screen size
- CPU and GPU abilities
- Format/codec support

Since the session with the other endpoint is never stopped, and since the other endpoint is blissfully unaware of the preceding migration, it will continue to send application data in a format suited to the *source* device. Thus, there may be need for adaptation of this content before it is sent to the target device. Similarly, there may be a need for adaptation of the content sent *from* the target device as well. E.g. in the case of a videochat session, to avoid content loss, this content adaptation has to be performed in *real*-time. To perform the appropriate content adaptation, the proxy needs to keep track of context information regarding the connected devices and their applications, i.e. their capabilities and properties, supported protocols, formats, codecs etc. We argue content adaptation mainly will be used for transcoding between media formats, hence the proxy will have to implement a codec library, as well as mux (multiplexing) and demux (demultiplexing) libraries, which supports transcoding between the various media formats. Figure 11 shows a simplified sequence diagram where a proxy performs content adaption following a migration.
In appendix, section 12.2, we investigate the feasibility of content adaptation in a proxy-based approach. We concluded that, although it may be possible in some cases, it is not desirable to have a proxy perform content adaptation, due to, among other, the following reasons:

- Even though performing real-time content adaptation may be possible, by e.g. utilizing cloud technology and a large mux, demux and codec library, the content won't necessarily be optimally suited to its new context. E.g. in the case of a video streaming where the client endpoint is a mobile phone (connected to a mobile cellular network), the video quality (and format negotiated) used may be low. When the session is migrated to e.g. a Smart TV, since the proxy only receives this low quality video, it can't be upgraded to HD, which will result in a poor video quality.
- Potential big and complex overhead at the proxy, as it has to keep track of all the needed context information in order to perform the appropriate content adaptation
- Potential bottleneck and scaling issues, considering a proxy performing real-time content adaptation on multiple concurrent application sessions
- The proxy may have to implement server-like functionality if it is to provide clients with the adaptable/scalable features originally provided by an application server

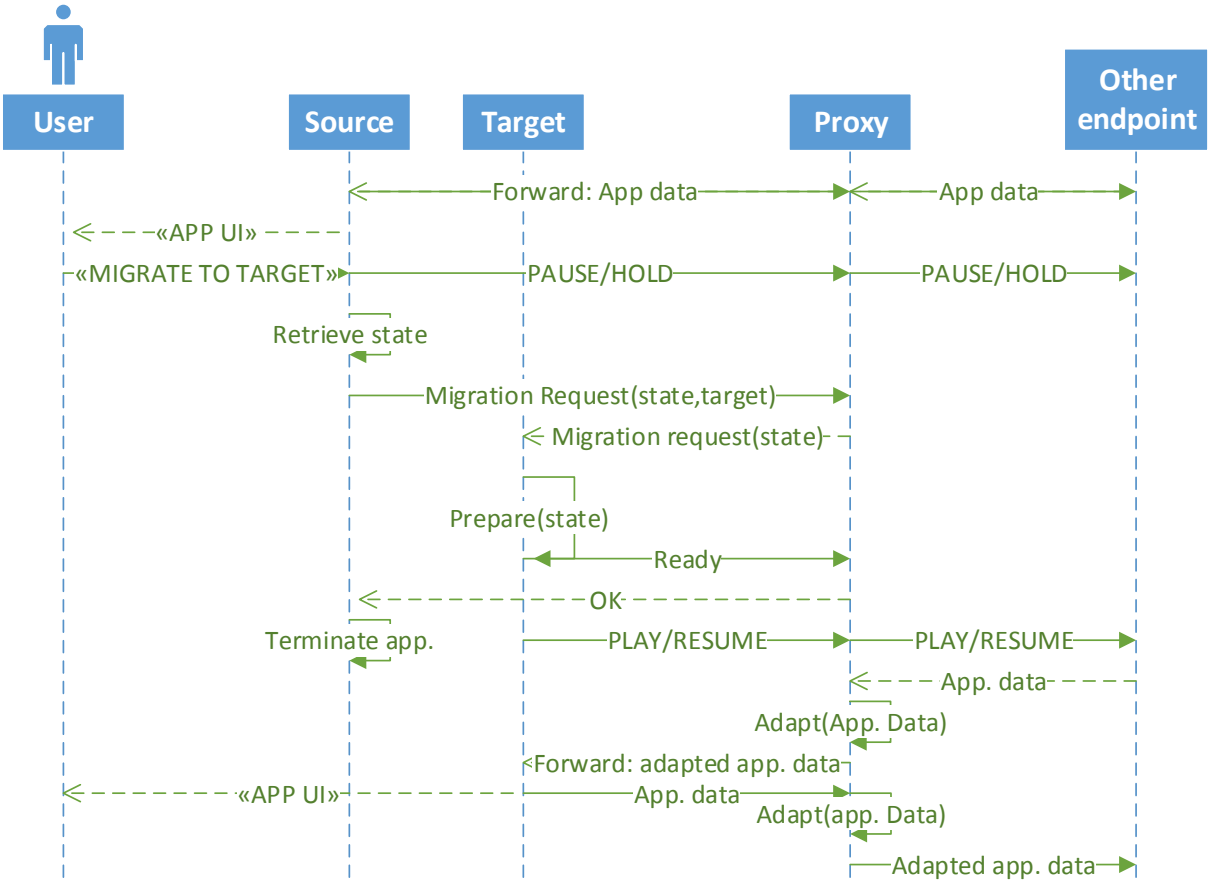

Figure 11. A simplified, proxy-based live migration approach

Evaluation

The proxy-based live migration approach we've presented should be able to provide cross-device session mobility for any type of application. By implementing the migration functions in (or near to) the proxy, we both avoid substantial application modification, as well as minimize the complexity and workload done on the devices. Though, following a transparent migration, the proxy may have to perform real-time content adaptation, which will require it to have sufficient context information about the users, devices and applications it serves. This requires a high-performance proxy with a potential huge overhead, and may still cause a non-optimal user experience from the target device. Thus, the proxy-based live migration approach remains a potential option, but with an undesirable content adaptation element.

When considering the various application categories and types the solution is to work with, the goal of retaining the original session (i.e. the live migration approach) arguably only strongly applies to real-time communication applications, where we have to consider the user experience of a remote user. In the case of non-real-time applications (e.g. a video streaming application), however, we don't need to account for this. In those applications, a simple workaround can be to avoid content adaptation altogether, by e.g. stopping the session on the source device, and continuing it from the target device via a new request. This new request should specify the state to start from (e.g. the position in

a video streaming playback). This is actually the approach of a cold migration. This way, the target device and the other endpoint will negotiate and establish the most suitable session parameters, hence no content adaptation will be needed. We will discuss and evaluate the cold migration approach in the next section, and especially its viability when it comes to working with RTC-apps.

6.2.3 Cold migration approach

Cold Migration is the case where the application is stopped and then started again. Under this scenario, the user can experience some downtime that occurs from stopping a session and practically establishing a new one. Though, arguments should be provided upon launch, ensuring that the "new" session starts where the previous one was left off.

In a cold migration approach, we don't require the migration to be kept transparent to the other endpoint. Thus, we avoid the need for a proxy-like entity, and consequently we avoid the need for content adaptation. Whenever an application session is migrated, a completely new session establishment between the target device application and the other endpoint is established, and the appropriate session parameters are negotiated. After the user requests a migration from one device to another, the source device application should retrieve its session's state and transmit it to the target device. The target device then launches the requested application by passing this state as an argument, ensuring a "stateful" launch. This should provide a seamless migration from the end user's point of view, which we argue is the most important thing to consider. See Figure 12.

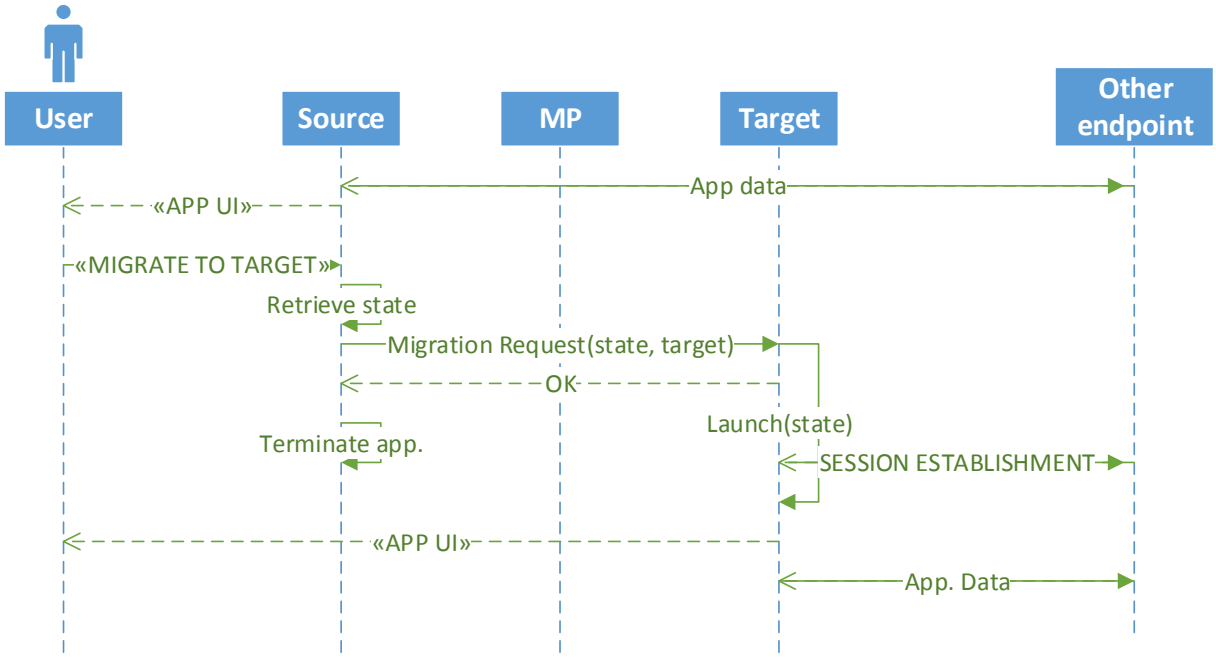

Figure 12. A simplified cold migration approach. Here, the MP is a potential Migration Platform entity. Messages may or may not go via this entity, depending on the architecture of the solution.

RTC-consideration

However, we have to consider how such a migration approach may be perceived at the other endpoint as well. In the case of real-time communication (RTC) applications there's usually a person at the other endpoint, not an application server. We need to consider that person's point of view when performing a migration as well.

Consider e.g. a videochat session with a remote peer/user. Now, we want to migrate the videochat session to another target device. If we simply stop the application on the source device and start it again on the target device, the remote person will perceive it as you first hang up and then call him/her up again. This is an undesirable behavior. Thus, we suggest the following workaround for RTC-apps:

- Before an upcoming migration, the application sends an application-specific *notification message* to the remote peer/application
- The remote application/peer receives the notification, where it is made aware of the upcoming migration, and can consequently handle it appropriately, e.g. by communicating it to the user and entering a "wait" state, waiting for a new call from the migrating peer.
- Once the remote application has responded appropriately, it sends a message acknowledging the upcoming migration. When this is received by the migrating peer, a normal cold migration procedure is executed.
- The "stateful" launch in this case could be having the target device application calling up the remote peer. When receiving this call, the remote peer validates

that the call is from the same user, accepts the call, and a normal media session negotiation and establishment is performed, before the videochat session is resumed.

In order to avoid too much session disruption, the above should occur without much delay. The workaround will require some additional application modification, but we argue it is minor, as there's likely already a communication channel established between the peers over which the notification message can be sent.

Evaluation
The cold migration approach we've presented should be able to provide cross-device session mobility for any application, though with the need for requiring an additional, but minor, application modification on RTC-applications. It is based on a relatively simple scheme, where the application session is terminated on the source device, and started again on the target device. As such, there is no need for introducing complex entities, and the solution should be fairly lightweight.

6.2.4 Migration conclusion

After examining the possibilities of both live and cold migration, we have seen the benefits of each approach, in conjunction with the difficulty and complexity of its implementation. We've considered a proxy-based live migration approach and a cold migration approach, where the first retains the original session throughout a migration, while the latter terminates the application session from the source device, before it's resumed from the target device. Both are dependent on some sort of transfer mechanism, through which the source device application can send a session/state depiction to be passed to the target device application. In the case of live migration, this will help prepare the application to resume the ongoing session, while in case of a cold migration, it will enable the application to perform a "stateful launch".

In the proxy-based live migration approach, all application data goes through a proxy entity, which, if needed to, performs real-time content adaptation. This creates a rather complex solution, with high performance requirements on the proxy. We've argued that the live migration approach only is justifiable in the case of RTC-applications, while in all other cases, a cold migration approach is just as good, if not better, as we don't have to consider a remote user's experience. Hence, as building a solution based on adapting, proxying and forwarding data between devices will increase the complexity and deployment cost of a solution without offering significant benefits, we argue a cold migration approach is the best option for a cross-device session mobility platform. For the scope of our solution, cold migration provides us with the same functionality that live does – a seamless session migration from the user's point of view – with the

exception of RTC applications, for which we've suggested a workaround, requiring a relatively minor additional modification. This is why we conclude that implementing our session migration with a cold migration scheme (stop, transfer and start) is our best option.

The next thing we need to consider is the architecture actually supporting the cold migration solution. I.e. an overall architecture supporting both device discovery and session transfer. What entities are needed, what are they responsibilities, and where will the complexity reside?

6.3 Architecture

In the following subsections, we are going to discuss the general architecture of our solution. Before a migration can take place, a way for registering the user, the device and the various applications must be established. It should be possible to discover other devices currently associated with the platform, and it should provide a way of communicating migration requests, as well as provide means for transmitting the session/state depiction object(s), between the devices. In addition, we need to decide where the logic of our solution (e.g. the components realizing the migration functions) is going to reside. We will analyze our options and try to choose the best architecture supporting the chosen cold migration scheme.

As mentioned in in chapter 4, an architecture for session mobility can be either *network-centric* or *device-centric*. The main difference between these architectures is where the migration functions, and thus the complexity of the system, reside. In a *network-centric* approach, the complexity is moved away from the end devices, towards entities that reside in the network, while in a *device-centric* approach, the logic and complexity is implemented mainly in the end devices.

In the following sub sections, we will evaluate both approaches, before we make a choice of architecture.

6.3.1 Network-centric

In a network-centric approach, the migration is at large handled by the network. The role of the user and the device is restricted to minor tasks, like advertising themselves and implementing interfaces to the migration functions in the network. Thus, according to *Movable-multimedia: session mobility in ubiquitous computing ecosystems* (44), it is associated with vast complexity within the network and high deployment costs. Therefore they argue this approach may be most suitable for small, localized session-mobility-based systems, with regard to the complex and costly infrastructure.

A network centric approach will need a centralized entity – for example a server – where the users authenticate and register their devices and applications. This server will keep track of all the registered users, their devices, and the applications available on each device. In a pure server-approach, the server must be able to send a list of available devices when requested to do so, as well as forward appropriate migration

requests/responses between devices when prompted to do so. As such, everything goes via the server. This approach results in little complexity implemented on the device.

Keeping everything centralized can make the resource management and monitoring more effective, which is an advantage from many perspectives. However, as everything goes via the server, we have a single point of failure, making the system not very robust. We can overcome this problem by introducing backup servers, although this will result in a higher deployment cost. Additionally, adding users, devices and/or applications can induce in scaling problems, as it will both increase the complexity of user/device tracking and increase the general workload and bandwidth consumption.

The performance of a network centric approach is dependent on the actual location of the server/network entity. We will consider two alternative deployments of a network-centric migration platform; a globally available server and a private server for each LAN.

Global and LAN-based Migration Platforms

When talking about a global migration platform, we mean the case where one or more globally available servers are used, to which all users register to. While this approach minimizes the deployment costs for a server-based (network-centric) approach, it can lead to scaling issues depending on the amount of registered users. At the same time, having one centralized server introduces a single-point of failure. However, by e.g. utilizing cloud computing, this issue would not pose a serious problem.

A LAN-based migration platform will only be used by the owners of the LAN. At first thought, this should drastically reduce the complexity of user tracking, the workload and the bandwidth requirements on the server, compared to a global migration platform approach, as there likely won't be that many devices connected to it. The problem with this approach is that if we wish to implement cross-domain migrations as well (which is possible by collocating the migration platform with the Access Point (AP), giving it a global IP), the platform has to support device discovery in two different ways. One for devices inside the LAN and one for external devices. Additionally, by collocating the migration platform with the AP, it should ensure high performance within the LAN, but could produce delay when working cross-domain. That is why we believe that a global migration platform is a preferable alternative.

6.3.2 Device-centric (peer-to-peer based)

In a device-centric architecture, the mechanisms and functions for realizing session mobility are placed on the device, using the network as more or less an unintelligent infrastructure for data transfer. This vastly reduces network dependency, though it requires more capable and complex applications and devices.

When this kind of functionality is put on the end device, it minimizes the complexity on the server entity, or even eliminates the need for a server entity at all. The device-centric architectural scheme we will examine in this section is a peer-to-peer (P2P) based architecture for communication directly between devices.

A peer-to-peer based networking scheme enables peers to communicate directly to each other, unlike a client-server based scheme, where communication has to go via a

server. There are several existing P2P paradigms for device discovery and connection initiation between the peers, with varying implementational complexity. For use in our solution, one such appropriate paradigm will be used to which devices can connect/register to, enabling them to discover the available devices they can migrate an application session to. Upon a migration, a direct peer-to-peer connection (data/signaling channel) will then be established between the source and target device, over which e.g. the application session/state will be transmitted.

For our approach we will consider two alternative schemes; a centralized and a pure peer-to-peer scheme (57).

Centralized peer-to-peer

In a centralized peer-to-peer scheme, a central server is used for indexing functions and to bootstrap the entire system. Although this has similarities with a structured architecture (an architecture where peers get addresses and are inter-connected via a server), the connections between peers are not determined by any algorithm. In a typical P2P application, the main use of this server though is mostly associated with indexing (keeping track of which files lie in which peer) in a p2p network. This can be adapted in our case, to track other useful information, such as user- and device-associations.

Evaluation:

A Centralized P2P is less robust compared to other P2P schemes, due to the introduction of and heavy reliance on the bootstrap server, which leads to having a single point of failure. The benefits of this approach are that it provides easy peer discovery and registration and allows us to manage the peers as we see fit (e.g. with creating logical groups of peers), which is not the case in unstructured P2P schemes.

Pure peer-to-peer

In a pure peer-to-peer scheme, the entire network consists solely of equipotent peers, meaning peers that have “equal rights and equal responsibilities” inside the network. There is only one routing layer, as there are no preferred nodes with any special infrastructure function, which means that there are no “special” nodes. This category can be further divided in structured and unstructured p2p networks, but we will not go into a detailed analysis of the two.

Evaluation:

A Pure P2P scheme is more robust than the centralized P2P scheme, because of the lack of a centralized server. Instead, every peer acts as both client and server. This leads to slower discovery and registration/deregistration, because of the nature of overlay networks and the nature of overlay routing. Sometimes, when nodes become inactive or get disconnected, the discovery of this event can take time. In overlay networks every node acts as a “router” with its routing table being changed constantly and updating other nodes about its changes. So in the case of one node disconnecting, the neighboring nodes might be aware of this almost instantly, but others might need more time to discover it. This property can be a nuisance for users in our system. (58)

Pure vs Centralized P2P comparison

Since our solution will focus on small "logical" groups of peers (e.g. the devices owned by a user, or a household) and we wish for every peer to know the address of every other peer in its group, a centralized P2P design with a server maintaining address and grouping lists of peers would be the best option. This way, by following a centralized P2P architecture, we move complexity away from the end-device as much as possible and we keep the disadvantage of having a single point of failure at a minimum.

6.3.3 Architecture Conclusion

For the purpose of addressing, we will need a server either way, where the devices can register as we explained in previously. The central server will allocate/lease "logical" addresses (such as UIDs) to every peer after the user registers his device. Based on the provided information upon registration, the device will also join a group consisting of all the available and active devices registered to the same user.

In order for a node to know where it can migrate to, it has to maintain a list of addresses much like a routing table. Regardless of whether we choose a network-centric or device-centric architecture, a routing table will need to be kept and updated at the server.

Both architectures can provide the discovery and addressing mechanisms we need for the cold migration to work. In Table 4, we compare how a device-centric (centralized peer-to-peer) and a network-centric (client-server) approach perform in key areas in order to decide which scheme works best for our solution.

Centralized P2Pvs Client-Server	
Scalability	P2P networks are by default very scalable, which makes them the better candidate based on this criteria. A server-based solution has a limit to the number of clients it can serve at a time by default. This issue can be addressed by Cloud Computing and its automatic scaling mechanisms. Still, all things considered, P2P is a better option for scalable systems.
Robustness	Both approaches use a server, which means that we have a single point of failure. The advantage though, goes to P2P. This is because the server is used only for addressing and discovery, meaning that already existing peer connections, will not be affected by the failure, unlike a client-server scheme[1].
Deployment cost&complexity	In this regard, the deployment complexity is relatively straight-forward for both alternatives. A server used for addressing is needed for both approaches, while in the "client-server" approach the server will also implement the migration functions and logic, rendering it slightly more complex and costly.
Performance	A (centralized) P2P scheme would in theory have the same performance as a client-server scheme when it comes to peer discovery, since the "addressing" server performs the exact same functions in both cases. Regarding network performance, the P2P scheme avoids network bottlenecks by connecting peers directly to each other instead of sending all traffic via server, which makes it the better candidate.

Table 4. Solution architecture analysis

Based on our analysis, presented in Table 4, we argue a centralized P2P architecture works best for our solution, due to its advantage in key factors as scalability, robustness and performance. Given that mobile devices (which are the poorest devices, performance-wise) are able to run complex applications and are becoming more and more powerful as of today, we do not think that any devices will be excluded from using our solution if we choose to follow a P2P scheme.

[1] Here, *Cloud Computing can be utilized to mitigate this issue, by auto scaling and ensuring availability at any time given any demand. However, this will lead to a higher deployment cost.*

6.4 Design Conclusion

In this chapter we have analyzed and discussed the possible alternatives when it comes to how an actual migration should be performed, and then, on the basis of our choice, we've evaluated alternative architectures supporting the given migration approach.

When comparing live and cold migration, we've concluded that a cold migration approach constitutes the best option of the two. They provide more or less the same functionalities, except that in the case of RTC-apps, a cold migration approach requires an additional application modification. By avoiding migration transparency, we consequently avoid content adaptation, allowing the migration to be achieved in a much simpler and efficient way.

When discussing the architecture, we analyzed and compared the benefits and drawbacks of implementing the main logic of our solution on the server-side (network-centric) and on the client-side (device-centric). Based on a comparison of the two alternatives in areas such as performance, scalability and robustness, we saw that following a device centric paradigm would yield better performance in most areas for our solution.

To summarize, through this theoretical discussion and analysis we have concluded that our solution will provide session mobility based on cold migration approach. The main logic of our solution though, will reside on the end-devices which are going to communicate in a peer-to-peer fashion with the help of a centralized addressing server. In the next chapter, we will use these findings to realize and present our solution design.

7 Solution design

7.1 Introduction

In the previous chapter, we discussed and deducted the most suitable design aspects of a generic cross-device, cross-network session mobility platform for applications. In this chapter we build on the conclusions made in the previous chapter, and present our proposed solution design.

Our proposed solution, the Migration Platform (MP), will consist of two entities (see Figure 13):

- The server-based Migration Server(MS)
- The client-based Migration Client (MC)

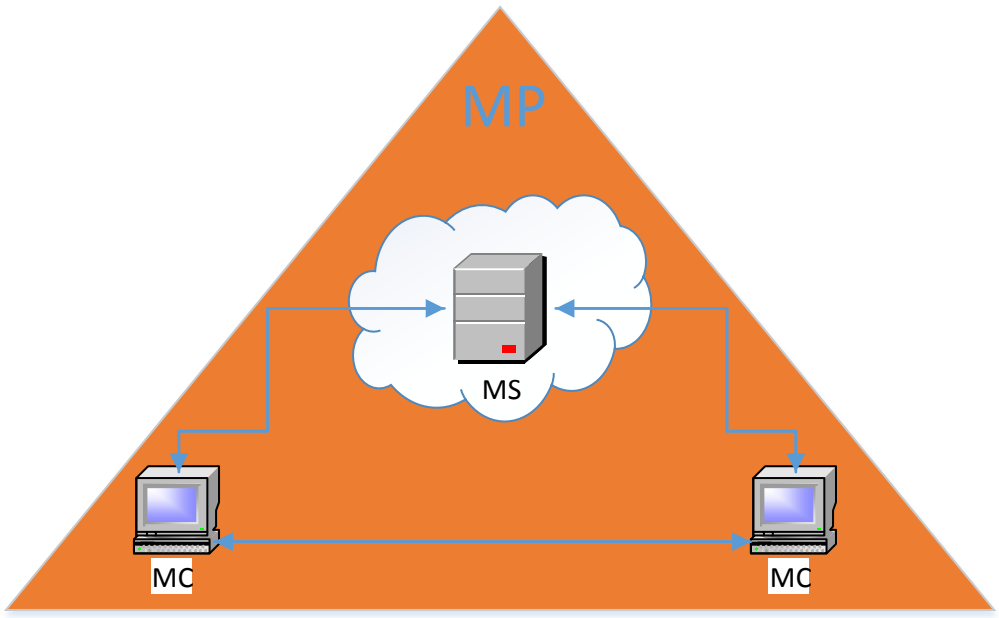

Figure 13. The Migration Platform (MP) and its two entities; the server-based Migration Server (MS) and the client-based Migration Client(MC).

While the MS's main responsibility will be to interconnect the various devices, the MC's main responsibility will be to provide and establish a platform for application state transfer between devices. Together, the MC and MS will enable an application session to be migrated between devices. The actual migration is done in a 'stop-and-start' fashion (cold migration), as described in the previous chapter.

Each device will have to install a device/platform-specific MC, which connects the device to the MS (and advertises it to the other connected devices). The applications that are to utilize the capabilities offered by the Migration Platform need to implement an interface to the MC. Thus, all messages related to migration will be done between the MS and MCs. Application-specific signaling will remain as before, i.e. the solution should not require any server-side modification. See Figure 14.

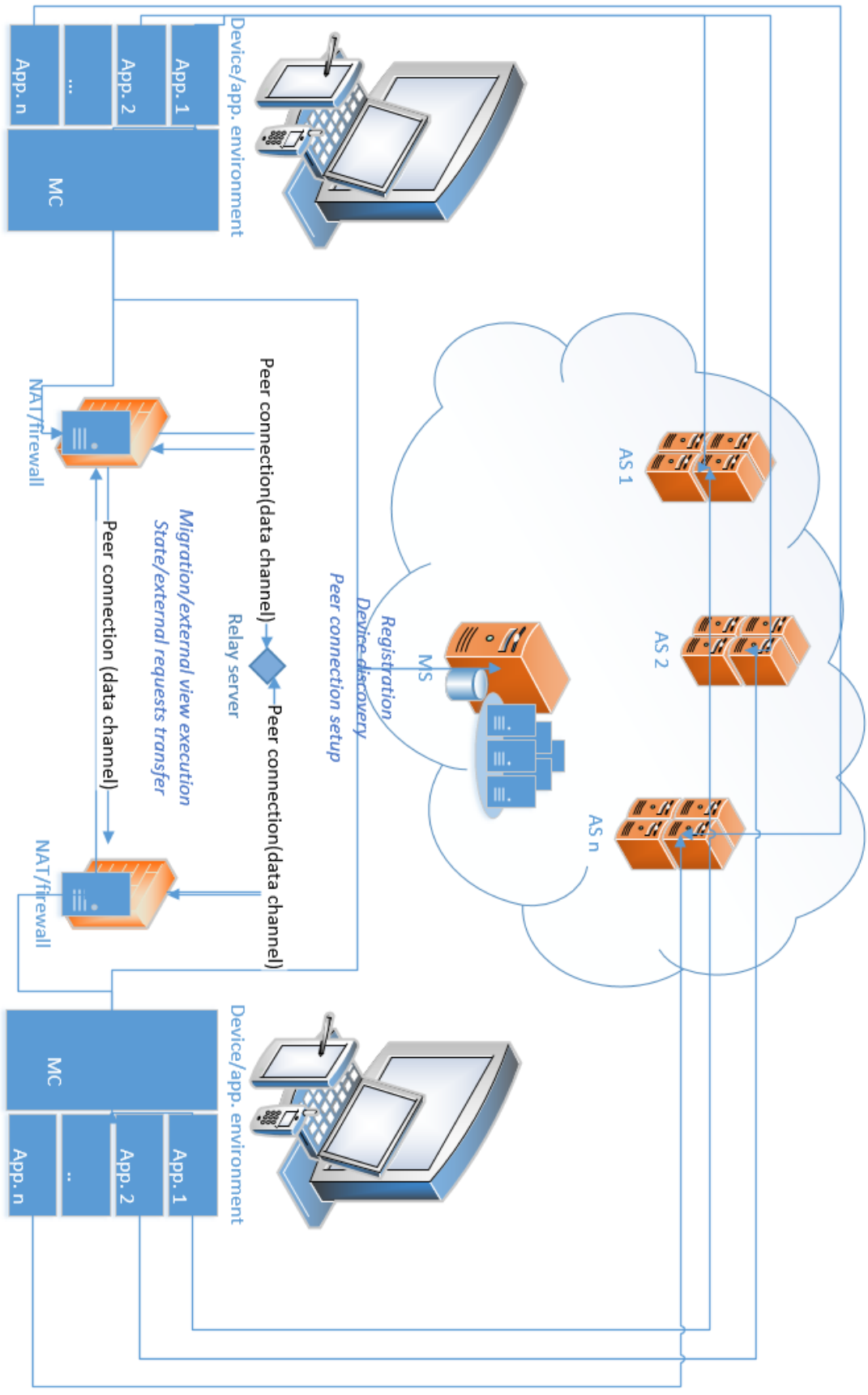

Figure 14. The solution architecture. Each device has a device-specific MC installed, with several applications interfacing to it. The MC connects to the MS and establishes direct data channels with other MCs upon migration.

In the following sub sections we will look at the MS and the MC (as well as its interface to the applications) in more detail. Lastly, we will show how the MP actually works, by displaying various sequence diagrams showing the message flows between the entities involved.

7.2 The Migration Server

As discussed in the previous chapter, we believe a centralized peer-to-peer architecture would be the best option for our solution. Thus, the Migration Server (MS) should act as a bootstrap server, indexing and creating associations dependent on the device's user id and applications.

The MS's main responsibilities will be to:

- Bootstrap the entire system, providing connectivity for the MCs. As many different devices, with different capabilities, are to connect to the same MS, the MS may have to be able to support multiple transport mechanisms in order for all devices to be able to connect to it.
- Keep track of associations (users, devices and applications)
- Aid in setting up peer connections between devices upon a migration

Deployment

The MS will reside on server-side, deployed in a cloud-like fashion, ensuring that the server has the capability to scale on-demand. As such, the server will be globally available, ensuring that the cross-domain mobility requirement will be met. Additionally, by implementing back-up server(s) we can avoid the single point of failure problem.

By utilizing the properties of Cloud Computing, mentioned in 3.2, we ensure that the server will be able to handle a continuously fluctuating amount of requests, and thus react immediately to a consumer's demand. This happens on-demand by pooling, scaling up and scaling down resources, optimizing resource use.

7.3 The Migration Client

The Migration Client (MC) is a client that is installed natively on every device. It connects the device to the Migration Server and enables users to migrate running applications between his/her devices.

We argue we should try to minimize the required user-interaction with the MC. The user should set his/her user credentials (and optional settings) upon installation of the MC, but the user shouldn't be required to interact with the MC after this, unless she wants to. Thus, the MC will run as a background service, transparent to the user. The only user interaction related to migration after this will go through the various applications' user interfaces.

The MC consists of four modules (see Figure 15):

- The *Signaling module*. Responsible for signaling and communicating with the MS.
- The *Peer Connection(PC)-module.* Responsible for establishing and maintaining peer connections used in migrations.
- The *Application-interface-module.* Responsible for providing an interface to the applications that are to use the service.
- The *Native module.* Responsible for interfacing to the device's hardware and launch applications.

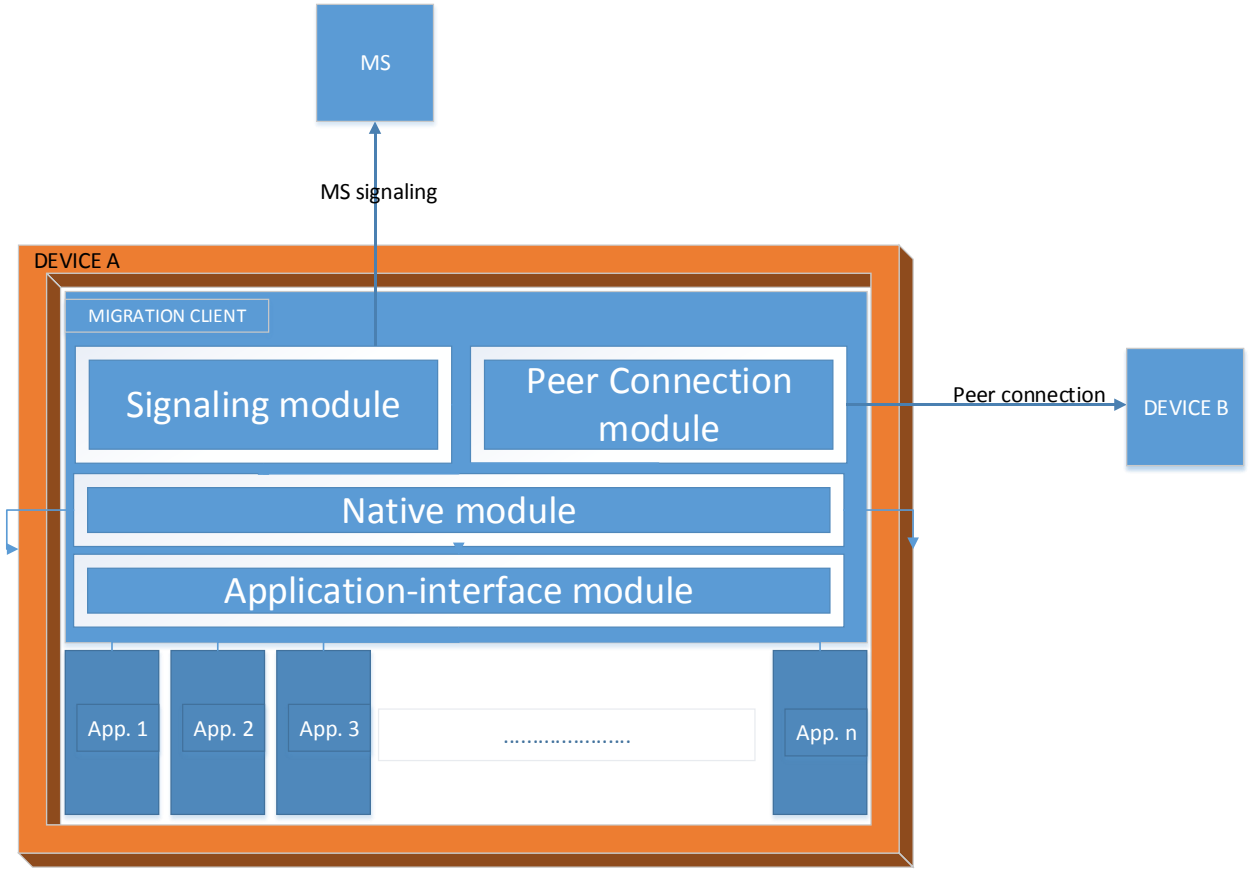

Figure 15. The Migration Client and its modules

We will now explain each module and its responsibilities in more detail.

The signaling module

The signaling module should be responsible for providing connectivity, connecting and registering the user, and advertising the device (and its applications) to the Migration Server. It should listen to both local requests from the various registered applications, and remote requests received from the MS. It will also help establishing peer connections for migration operations by sending migration offer/answer messages via the MS (i.e. before a direct path has been established).

The signaling module is responsible for continuously maintaining the device's connectivity (by sending so-called heartbeat messages to the MS on predetermined intervals) as long as the device has Internet connectivity. The MC should also be able to maintain the device's connection to the MS upon terminal mobility, i.e. a change in access network.

Finally, if the user installs/registers (or uninstall/unregisters) a "migratory" application, this update should be communicated to the MS - and consequently all other connected devices registered to the same user.

The Peer-Connection module

The Peer-Connection (PC) module is responsible for setting up and maintaining a peer connection between two devices during a migration session. It should try to establish a direct data channel between two devices, where messages will be sent. If no such direct path can be established (e.g. due to NAT/firewall issues and/or device interoperability issues), the data will be sent via a relay server. We differentiate between two different migration possibilities:

1. Full migration
2. External view

In a *full migration*, the entire session is to be transferred from the source to the target device. After a successful state transfer, the peer connection, as well as the source device's application, should be terminated. The application should then be launched (in a stateful manner) on the target device, resuming the session from the point where it was left off in the source device.

In an *external view* session, however, the peer connection must remain open even after the state is transferred and the application is launched on the target device. In this scenario, the source device will act as a *controlling* device whereas the target device will act as a *controlled* device. Hence, the user will be able to control the view on the target device from the source device (e.g. controlling the TV from the smartphone).

The Application-interface module.

The application-interface module is responsible for

- Registering an application to the Migration Client
- Providing the interface that the applications have to implement

Application registration

First of all, the application needs to register itself to the Migration Platform. This can be done e.g. during installation of the application, or later. Each unique application needs a unique identifier in order to differentiate between them. Those unique identifiers should be kept in an application registry at the MP, much like DIAL's Application Name registry (59). This will ensure that the identifier for each application is well-defined, hence avoiding any naming/identifier conflicts. The registry should be issued and maintained by the MP, and made available to anyone implementing or using the service.

Application interface

For an application to be able to utilize the migration capabilities provided by the Migration Platform, the application has to implement an interface to the MC. The application-interface module gives the application access to all the migration functions provided by the Migration Platform. However, it is required that the application developers implement a couple of application-specific functions. There are two essential, mandatory functions that need to be implemented:

- *State retrieval* – upon a migration, the application on the source device needs to retrieve its current state to be sent to the target device. How the state should be described is completely up to the application developer. The only requirement is that the state is serialized, so that it can be sent over a data channel. If need be, it can be sent over several messages, i.e. it can be any size the application developer seems fit.
- *Stateful launch* – the application should be able to launch upon the received state argument, resulting in the application resuming the application session from where it left off on the target device. Two optional parameters, Immediate and ExternalView, will also decide how the application should be launched. If immediate is true, the application is started immediately. If it is false, it starts in a paused mode. If ExternalView is true, an external view session is established. If it is false, a full migration is performed. See Table 5 for the detailed explanation of a stateful launch.

Optional launch parameters		**Immediate**	
		True	False
External View	True	Immediate, external launch. The application should enter a controlled view and resume immediately.	Paused, external launch. The application should enter a controlled view in a paused mode
	False (i.e. perform a full migration)	Immediate launch. The application should resume the application session immediately.	Paused launch. The application should resume the application session in a paused mode.

Table 5. Launch parameters

Additionally, the application should provide a dynamic user interface with respect to the "migration state", such as the following:

- A "migration GUI" showing the available migration controls and the available devices, if any.
- Appropriate feedbacks upon a migration request trigger.
- Appropriate views (and controls) for the controlling and the controlled devices (for external view sessions).

How exactly this "migration GUI" is to be implemented is up to the application developers as they see fit.

Non-real-time vs. real-time applications consideration

As discussed, we need to take into account the different nature between non-real-time applications (e.g. streaming and browsing applications) and real-time applications (e.g. videochat and real-time gaming applications). While non-real-time applications easily can be stopped on one device and started on the other, the same isn't necessarily true for real-time applications. Here, we need to take into account the remote user at the other endpoint of the application session. If we suddenly stop such a session without giving a warning, that person is likely to interpret it as a session interruption, or an error. Thus, we suggest that the relevant real-time applications should send a message notifying the other endpoint about an upcoming migration. Consequently, the application can notify the person and go into a "wait" state until the session is resumed from the new target

device. We argue this is a rather simple, but very efficient way of ensuring a seamless migration in a real-time scenario.

The native module.

The native module is the module in the MC that is responsible for interfacing to the device's hardware. Upon reception of an application state, it is the native module's responsibility to launch the correct application, providing the state as an argument.

Why native?

In order to automatically launch applications, the MC needs to be able to interface to the device's native features, information and hardware (60). Consequently, the MC will have to be native. As such, for each different application development platform (e.g. iOS and Android), the MC will require its own development process, as each application development platform has its own native programming language (e.g. Android uses Java, iOS uses Objective-C), and often provides standardized Software Development Kits (SDKs), development tools and user interface elements.

As a result, compared to non-native applications, the MC will be more expensive to develop and maintain, and will require maintenance of multiple code bases. However, there are many available tools and frameworks for cross-device/OS development that can help streamlining this process. At least, since the MC will serve the exact same purpose on every device, we argue the actual development of the various native MCs will follow very similar design patterns.

Additionally, it is an advantage having the MC being native when you want it to run as a background service. Then you can just auto-start the MC at login/boot, and the MC will be active and listen for incoming requests without any required user interaction.

Automatically launch

Upon reception of the state data, the target device should automatically launch the given application, passing the state data as an additional launch argument. This is done by invoking the application via e.g. the device's native terminal, passing the state as an additional argument. The state launch, is, as stated previously, application specific, and thus out of the scope of our solution.

7.4 Call flows

Here we present a more or less framework showing how the solution should work in a generic and abstract manner, i.e. on any device. When using and/or implementing the MC on a specific device, there will be device-specific implementations that differ from one another. These device-specific parts will become clearer when we present similar call flows for our proof of concept in the next chapter.

In the following subsections we will present how the solution should work, by showing the flow of messages between the different entities involved. First, we will look at the client-server interaction, i.e. how a device and its applications connects (and stays connected) to the service. Then we'll look at the message flows in actual migrations between two devices, both full migrations and external view migrations.

7.4.1 Client-server and application connection/disconnection

When installed, the Migration Client should run as a background service on the device, automatically connecting to the Migration Server upon boot (given that the device has Internet connectivity). In order to migrate between devices, their respective MCs must be associated with the same user id (e.g. a unique user name). Additionally, in order for a user to be able to differentiate between his/her devices, each device should be given a unique name (e.g. 'smartphone', 'laptop', 'TV' etc.). Hence, upon connecting to the MS, the MC should inform the MS about its associated user id, its device name, and its installed applications. This way, the MS is able to keep track of its association groups, and will know who to inform upon connection, disconnection and other events. This results in MCs always having the most up-to-date association list.

Figure 16 shows what happens when a device connects/disconnects to the migration server, i.e. how a device advertises itself and how device discovery is done.

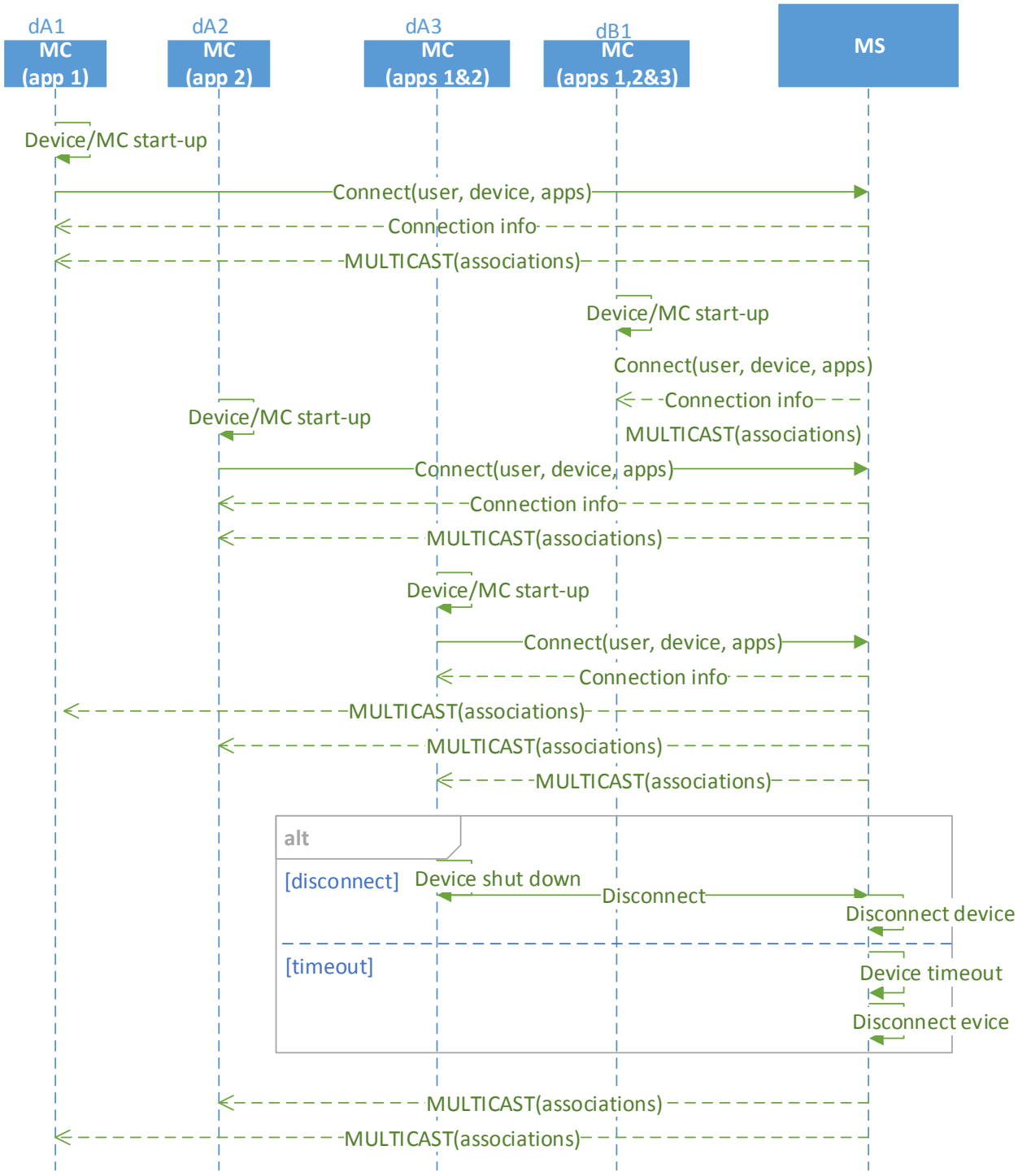

Figure 16. Device connection/disconnection and device discovery

Assume that there's initially no devices connected to the MS, i.e. dA1 becomes the first connection. Devices dA1, dA2 and dA3 belong to user A, while device dB1 belongs to user B. The devices currently have the migration-enabled applications shown in parenthesis installed.

As we see, upon boot, the MC in dA1 connects to the MS with User A's user id, a device id and a list of applications. This gets stored in the MS. The MS responds with connection information, e.g. a connection identifier that serves as a unique id for this device. Since there's no other connected devices at this time, the MS will also respond with an empty list of associations, via a multicast message. That way, the MC knows it can't migrate to any other devices at this time.

The same happens when dA2 and dB1 connects to the MS. Naturally, dB1's connection is not communicated to User A's devices, as they are registered to different users. But as we see, even dA2's connection is not communicated to dA1 as they don't have any of the same applications installed. This first happens when dA3 connects to the MS. Device dA3 has both applications 1&2 installed, i.e. both dA1 and dA2 can migrate to it. Consequently, the MS sends a multicast message to dA1 and dA2 informing about dA3's connection, as well as an association list containing devices dA1 and dA2 to the MC in dA3.

As with new connections, associated devices must be informed whenever one of their associated devices *disconnects*. A disconnection can happen in two ways; either a normal disconnect (i.e. a device shutdown or a manual MC disconnect), or by a timeout (e.g. as a result from a client crash or from losing Internet connection). Both alternatives are shown in Figure 16.

Heartbeat timeout

To ensure that the connected devices are available and active, the MC should send so-called heartbeat messages (61). This enables the MS to identify if and when the MC fails or is no longer available. See Figure 17.

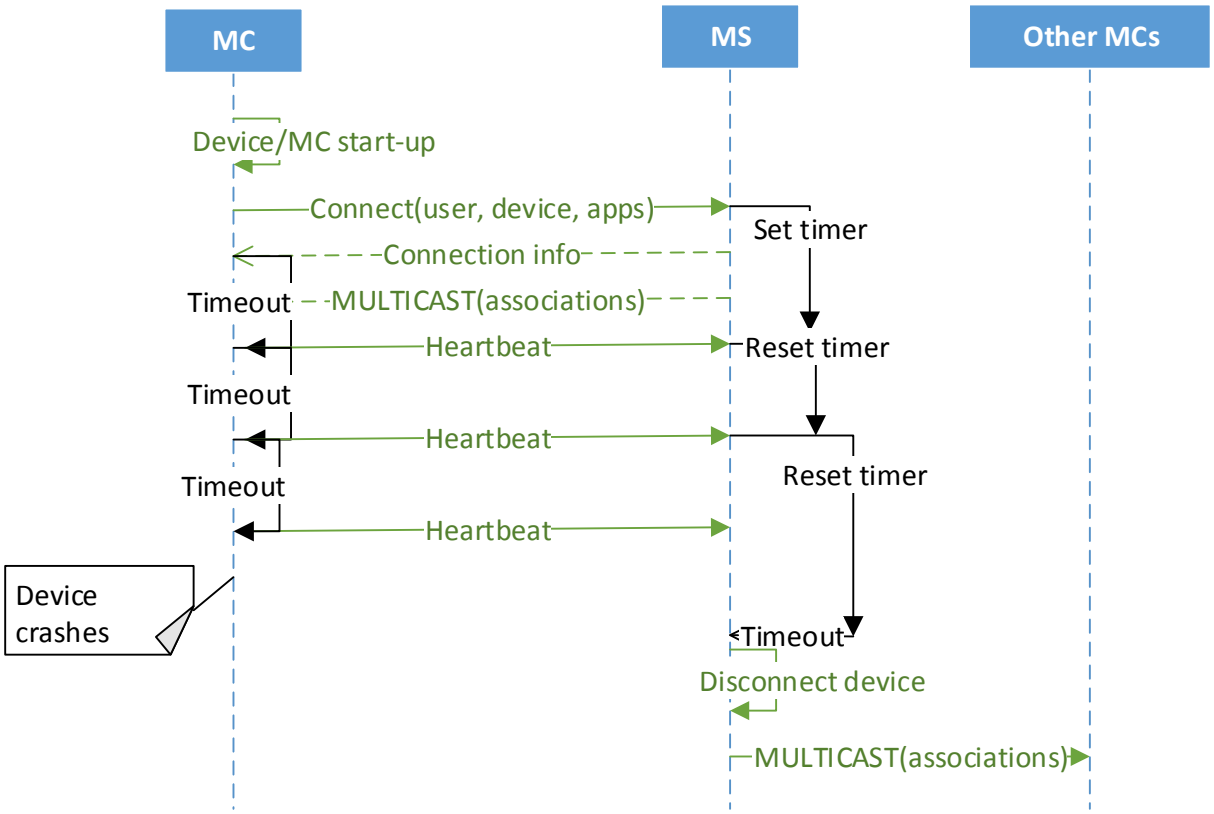

Figure 17. Heartbeat messages and timeout

We see that the MC sends heartbeat messages at a predetermined interval, indicating that it's still around and active. For every time the MS receives a heartbeat, it resets a timer. Thus, the MC's heartbeat interval should be shorter than the MS's timeout timer. If, for any reason, MS's timer times out, the MS interprets it as a client disconnection. It communicates the disconnection to the disconnected device's associated devices, and removes the MC from its list of connections.

Application registration

Since the MC serves multiple applications, it needs to be able to continuously register/deregister applications. Whenever an application implementing the MC-interface is installed on the device, this should be communicated to the MC's associated devices. Figure 18 shows such an application registration scenario.

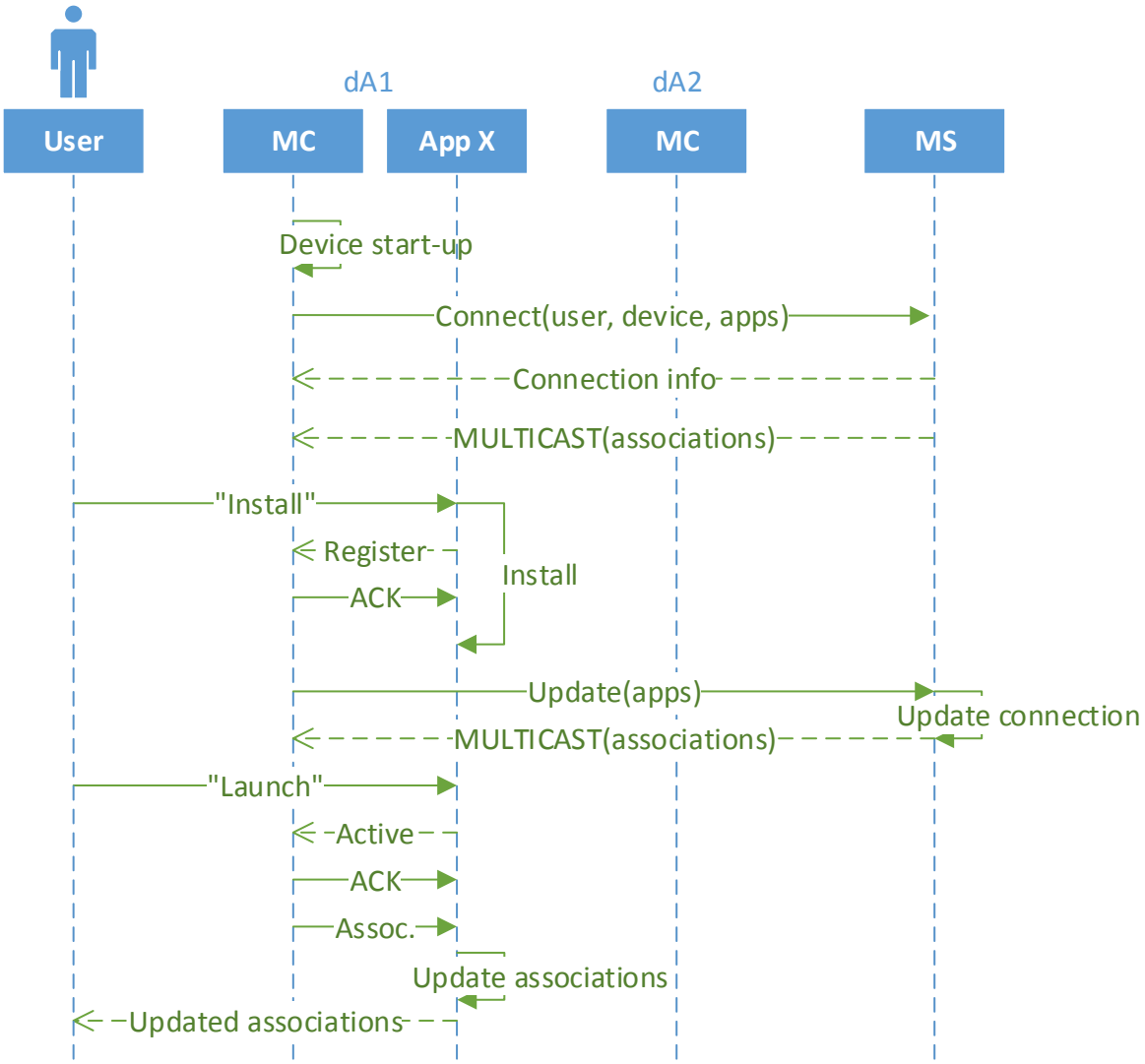

Figure 18. Application registration and application-MC communication

Here, devices dA1 and dA2 are associated to the same user, User A. Assume device dA2 already has App. X installed. Upon installation of App. X at device dA1, we see that the application registers to the MC. When the installation is done, the MC updates its list of applications to the MS, which communicates the new association between dA1 and dA2 to the two MC's. Similarly, the opposite should happen whenever a user decides to uninstall a "migration-application".

We also see that when the application becomes active, it receives an association list from the MC. This way, the user has access to an updated (app-specific) association list via the application's UI (up to the application developer), telling which, if any, devices the user can migrate to.

7.4.2 Migration negotiation

Now we know how the devices advertise themselves and how they discover each other. Via an application's UI, a user should be able to trigger a migration to another device, if available. This migration request can either be to do a full migration, or to establish an external view session between the devices, as previously explained. Additionally, the user should be able to decide whether the application on the target device should launch immediately or in a paused mode.

Essential in such a migration is the state transfer. The application on the source device has to retrieve its current state and pass it to the MC to be transferred to the target device's MC, consequently resulting in a stateful launch on the target device.

The actual migration, i.e. the state transfer (and a potential external view session), is done in a peer-to-peer manner, i.e. over a direct data channel. Before this, however, we need to establish such a peer connection. First, it is likely the devices are heterogeneous, and as such, they will have to negotiate and agree to a common data channel format. Secondly, the devices can possibly be located behind one or several firewall(s) and/or NAT(s), and as such they will have to find a way of traversing it/them to establish a direct path. If no such path is possible, the data channel will have to go via a relay server (see Figure 14).

Once a peer connection is established, the MC's can communicate to each other in a peer-to-peer manner, i.e. not via the MS. If the request was for a full migration, the peer connection gets closed after the state is successfully received. Else, if the request was for an external view migration, the peer connection is kept open, as the source (or controlling) device now is to send requests to the target (or controlled) device.

We will now look at two migration scenarios; a full migration scenario and an external view scenario.

Full migration

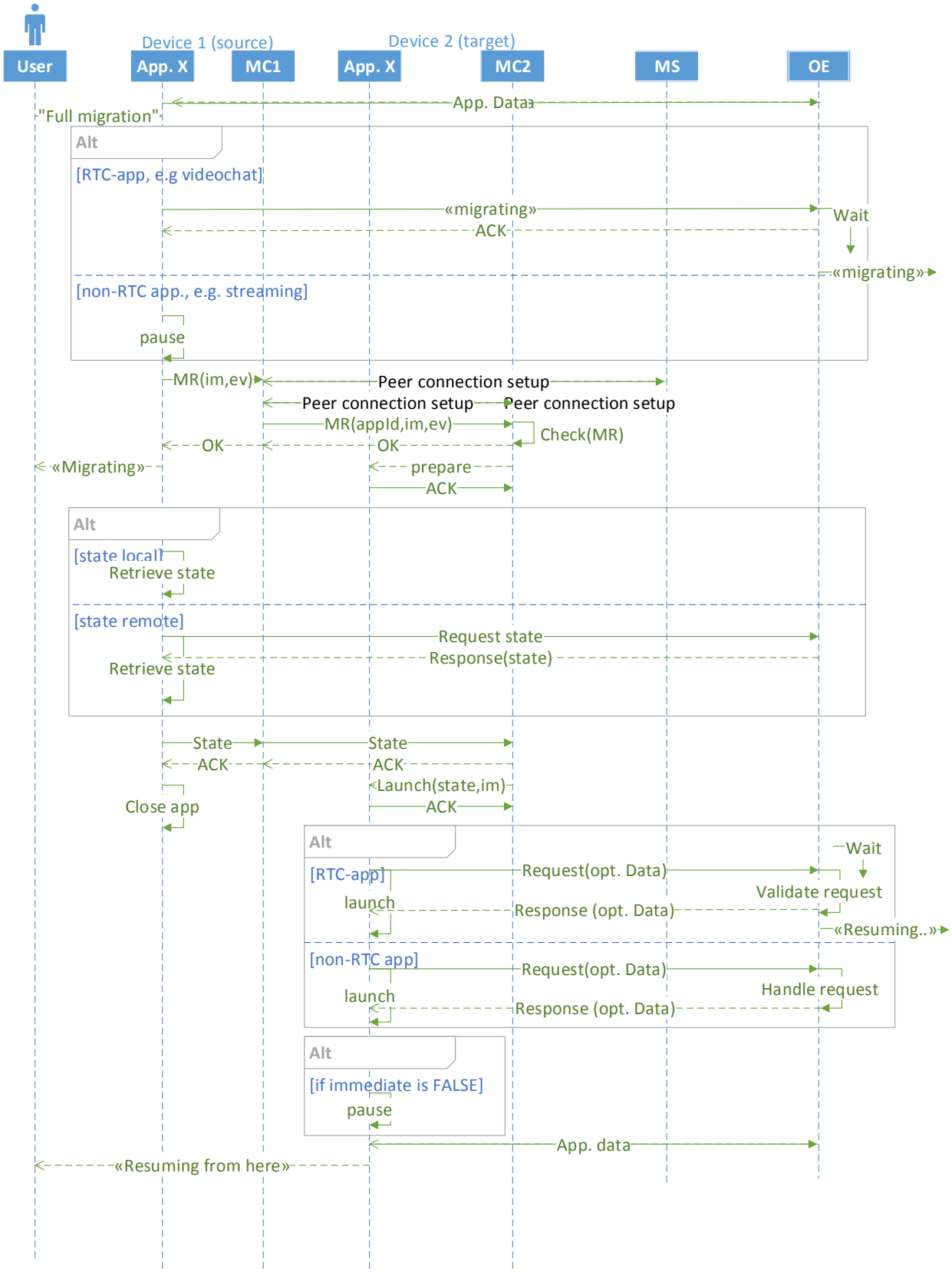

Figure 19. A successful full migration of App. X from source device to target device.

Figure 19 shows a scenario in which a full migration of a *running* application, "App. X", from source device 1 to target device 2 is being executed. Here, both devices are registered to the MS with the same user.

We see that at the time the migration is triggered, App. X is in an application session with "OE" (other endpoint). The OE can be either an application server (e.g. video streaming server) or a remote peer (e.g. videochat peer). Dependent on the type of the application, i.e. non-RTC or RTC-app, the user's migration request should be handled differently, as previously mentioned. We see that in the case of a RTC-app, the applications sends a "migrating" message to the remote peer to notify about an upcoming migration, whereas in the case of a non-RTC-app, the application is simply paused (we don't need to take into account any remote user's experience). We see that the remote RTC-app enters a "wait" state upon reception of the "migrating"-message. Here, the application sets a timer and waits for a new request (from the target device) to continue the session. During this time, the application ignores other calls, and only waits for a call from the same user to continue the session. If the timer goes out, the application will go back to a "normal" state. What we described here is an example of how a migration can be handled by an RTC-app.

When the MC receives the migration request, specifying the wanted target device and application (and Boolean parameters 'imm' and 'ev' corresponding to 'immediate' and 'external view'), it establishes a peer connection between the devices, with initial setup help via the MS. Once a peer connection is established, which also acts as an acknowledgment from the target device to the source device's migration request, the state can be transferred over the data channel.

Before the state can be sent, it has to be retrieved by the source application. The source MC requests the application to retrieve the state, and, dependent on the whereabouts of the state (local or remote), the application handles this state retrieval locally and/or with remote calls. Remember, this is app-specific, and up to the application developer to implement. Either way, the application returns the state (as a serialized object) to the MC, which sends it over the data channel to the target MC. Upon reception of the state from the application, the MC acknowledges it, and the application terminates on the source device.

Simultaneously, on the target device, the MC awaits the state object. Upon successful reception of the state, the target MC sends an acknowledge message back over the data channel, and invokes the application to launch, passing the state and the type of migration (immediate, external view). In this case, external view is set to false, i.e. the application should perform a normal launch, starting in an immediate or paused mode. Once again, we see the distinction between non-RTC and RTC-apps (even though in both cases, the application will eventually continue the session on the target device from where it left off on the source device). In both cases, a request is sent to resume a session (or start a new session, but from a certain point), but in the case of an RTC-app, the request is first validated by the remote peer to see that it's the same user it is waiting for.

Again, this is just a suggested way to perform migration-handling on an RTC-app. It is up to the app developer to implement such a feature.

Since this is a full migration, and no additional data is to be sent between the MC's, the peer connection is closed as soon as the source MC receives an ACK on the state.

External view

In the case of an external view, however, the peer connection will not be terminated after a successful state transfer and stateful launch. See Figure 20 and Figure 21.

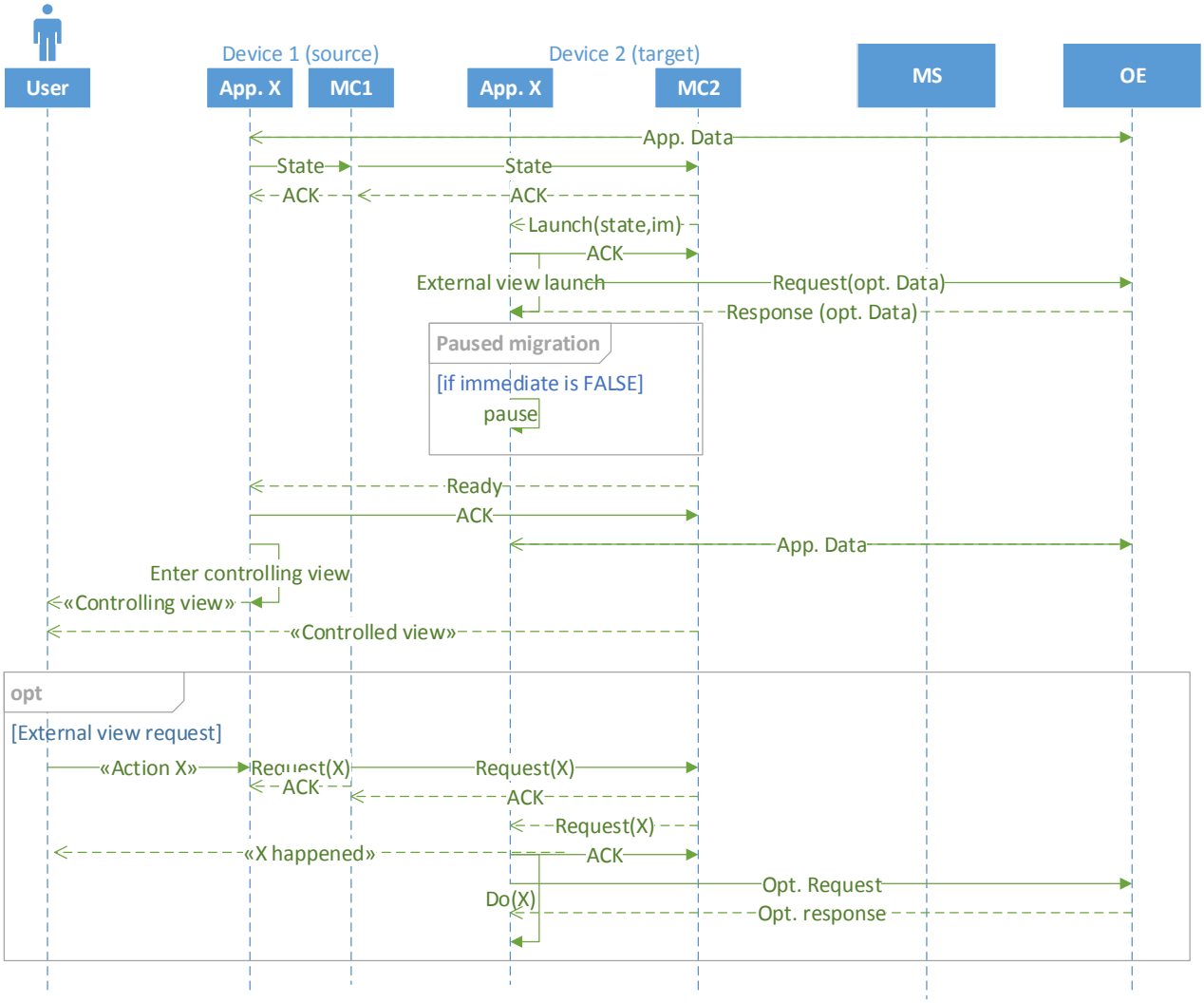

Figure 20. External view session establishment and requests

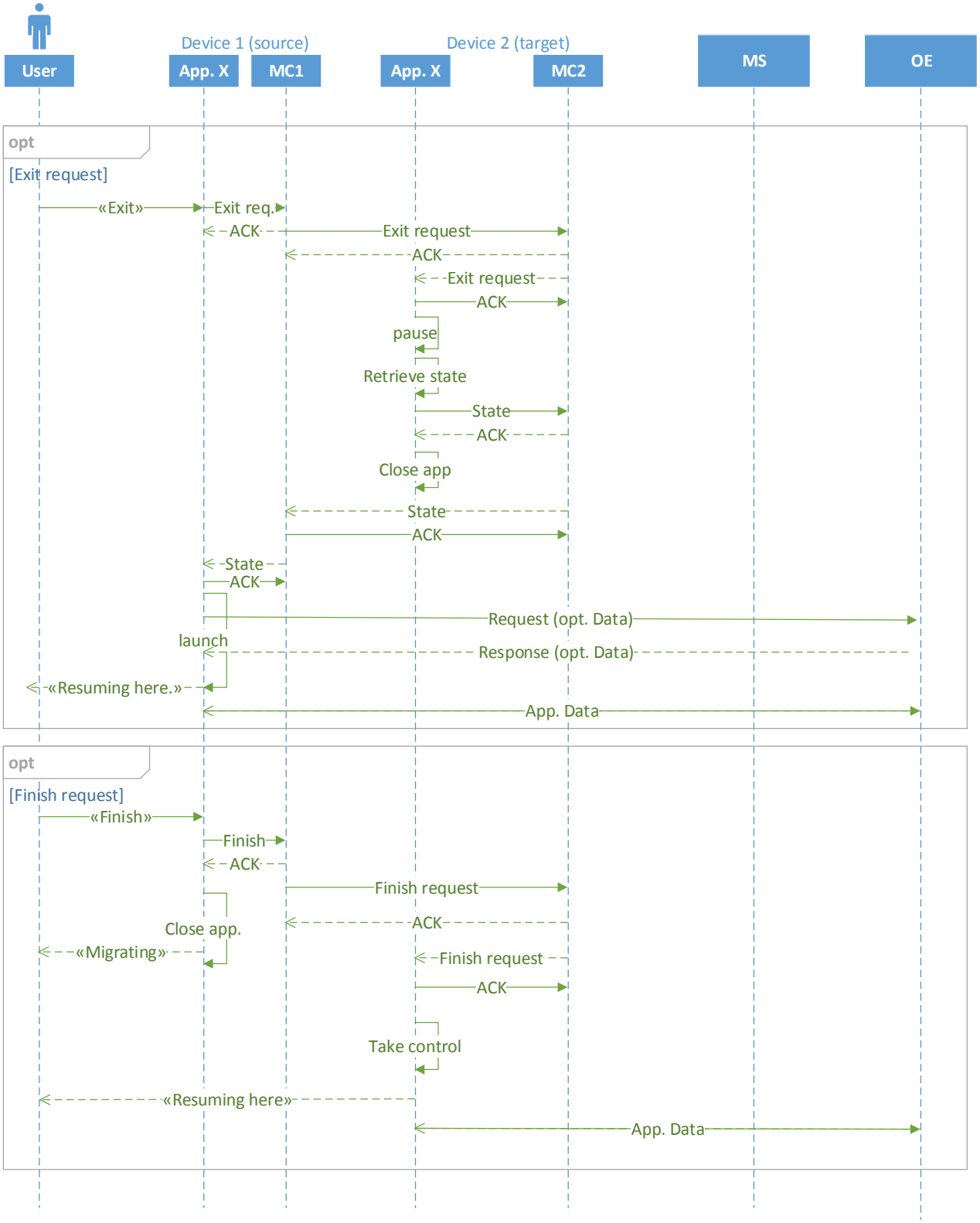

Figure 21. External view, finish and exit flow. Here, source and target are already in an external view session

Here, both devices are registered to the MS with the same user id. The user has requested an external view migration from source device to target device. Assume the target device's MC already has acknowledged the request, i.e. the peer connection between the devices is established (see Figure 19).

As with the full migration, the state gets transferred and gets acknowledged, but the source application is not closed, and the peer connection is not terminated. We see that the MC invokes the application to launch externally (i.e. to launch into a 'controlled' state). When the controlled application has launched externally, the MC sends a "ready"-message to the 'controlling' MC/application. The controlling application then enters a controlling view, which is displayed to the user. Now, the user is able to control the target device's application via external controls on the source device's application. From here, the user should be able to do the following:

1. Do external application requests (external view requests).
2. Exit the external view
3. Finish/complete the migration

External view requests are shown in Figure 20, while exit and finish flows are shown in Figure 21.

External view requests

External view requests are actually normal application requests, or actions, e.g. playback requests on a video player. By interacting with the controlling application's UI, the user can request actions that are to be executed on the controlled application (think e.g. using a smartphone as a remote control for the TV). The external view request is sent over the data channel via the MC's peer connection. On the target device, upon reception of an external view request, the MC simply forwards the request to the application, which executes the wanted action ("Do(X)").

Exiting the external view

The user should also be able to exit the external view at any time he/she pleases. With 'exiting', we mean retrieving the view on the controlling device. This also means transferring the state of the controlled application back to the controlling application, which means that exiting an external view session becomes very similar to a full migration procedure.

First, the controlling device sends an 'exit' request. The controlled device acknowledges the request, retrieves the state, stops the application, and sends the state back to the controlling device. The controlling device can then launch upon this state, and the peer connection, and thus the external view session, between the two devices is terminated. We're back to a 'normal view' on the target device.

Finish/complete migration

The user can also choose to go the other way, i.e. finishing (or completing) the migration to the controlled device. This is simply done by sending a "finish" request over the data channel (triggered by the user via the application's UI). We don't need to send any state, since the current state is already at the controlled device. Upon reception of the "finish" request, the target device's MC acknowledges the request, and notifies the application to 'take control'. The application on the target will then enter a 'normal view', resuming the session, while the source device's application gets closed. The user can now continue the application session from the target device.

Target application state handling

In order to avoid errors, or misconceptions of errors, during a migration, we need to take into account possible scenarios that can occur, and set rules that apply to each of them. Thus, Table 6 shows the actions the MC should do, depending on the current state of the application on the target device.

Application state	Action(s)
Not running	• Acknowledge migration request • Start application with provided state argument and launch parameters
Starting	• Send to source: "Application starting" • No action locally
Running	• Acknowledge migration request • Provide new argument(s) to application
Migrating	• Send to source: "Application already in a migration session" • No action locally

Table 6. MC's actions upon a migration request, given the application's state

Since the initial migration request from source to target device contains the application identifier, the MC can examine the state of the given application, and see whether it is available or not. I.e. it can decide whether the migration session (i.e. state transfer etc.) should continue or not.

A migration-application can take the following states:

- Not running.
- Starting. The application is just starting up.

- Running. The application is running normally, either in an idle or an active application session state.
- Migrating. The application is currently in a migration session, either migrating from, being migrated to, or in an external view session.

In most cases, we argue the application is either not running or running normally. In these cases, the application is available. To indicate this, the MC responds to the source with an acknowledgement, indicating that the peer connection can be established.

Otherwise, the application is either starting or already in a migration session. In these cases, we don't want to terminate those sessions. Rather, we respond with messages telling the source MC that the application is starting or already in a migration session. This should be communicated appropriately to the user as well.

Thus, in the cases where the MC responds with an error-message, the peer connection is not established between the devices, and the error message should be displayed to the user (or at least logged). Otherwise, the peer connection is established, and the migration continues normally.

By having the MC's sending heartbeat messages we're almost assured that the device we want to migrate to is available, but there is the possibility that the device has become unavailable during the short heartbeat time interval, i.e. before the MS's timer has timed out. Thus, unless the source MC receives a response to the migration request within a certain time, the MC will assume that the target device has become unavailable. This should be communicated to the user via the application. Similar handling can be done during an external view session.

7.5 Conclusion

In this chapter, we have proposed a solution providing a generic cross-device, cross-domain session mobility platform for "any" type of application, though requiring an additional, but minor, application modification on RTC-apps.

The solution is based on a centralized peer-to-peer architecture with two central entities; the Migration Server (MS) and the Migration Client (MC). All MCs connect to the MS, which aids in keeping track of associations as well as setting up peer connections upon migration. The MC is installed (natively) on every device, and provides an interface which applications wanting to utilize the migration functions have to implement. This requires application-specific implementation of certain functions, e.g. for retrieving the state, launching upon state, and for creating/executing external view requests/actions. Otherwise, the MCs and the MS provides and executes all migration functions. Upon a migration, the MCs create and establish a direct peer connection over which the state (with more) is sent. In an external view session this peer connection (data channel) is

kept open for sending/receiving of external view requests, while in the case of a full migration, the peer connection is closed after a successful migration.

In the next chapter we will present our proof of concept, showcasing the viability of the solution design presented in this chapter, by implementing a session mobility platform in the web environment.

8 Proof of concept

8.1 Introduction

Our proof of concept – the Web Migration Platform (WebMP) - leverages bleeding and cutting edge technologies to provide a generic, cross-device (PC/mobile), cross-domain session mobility platform for web applications and extensions.

The WebMP is heavily based on WebRTC (62), a free, open project supported by Google, Amazon and Opera, that "*enables web browsers with Real-Time Communication capabilities via simple Javascript APIs*". WebRTC is used in our proof of concept to create the Peer Connection module (see previous chapter). Here, we especially utilize WebRTC's RTCDataChannel API, which establishes a direct data channel between two browsers.

The WebMP is also based on a Socket.IO- (63) and Node (64)-based signaling part, where Socket.IO is used to create the Signaling Module, and both Socket.IO and Node are used to create the MS. These are both cutting edge technologies based purely on JavaScript (JS).

Thus, our JS-based WebMP, like the MP, consists of two entities (see Figure 22):

- The server-based Web Migration Server (WebMS)
- The client-based Web Migration Client (WebMC)

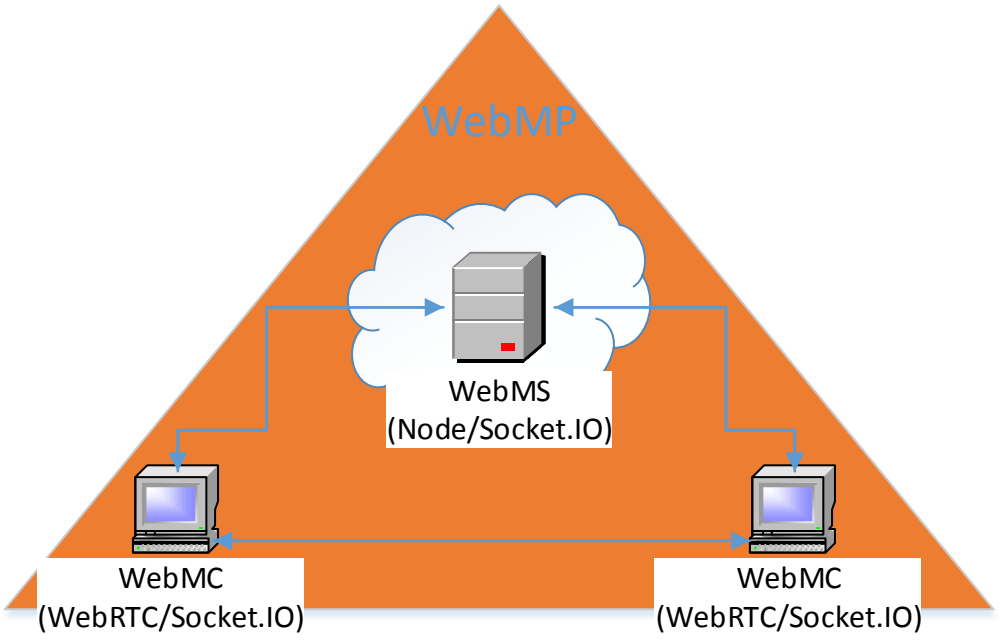

Figure 22. The proof of concept, the Web Migration Platform and its two entities; the Node/socket.io-based WebMS and WebRTC/socket.io-based WebMC.

The WebMC connects to the WebMS. Together, they enable a web application's session to migrate between devices. The WebMC is in essence a client-side JS-library which needs to be included by every web application that wants to utilize the migration features offered. This way, we *extend* the web application with migration capabilities.

We have created several demo applications that include the WebMC, as well as implement some application-specific methods, and thus are enabled to migrate. Specifically, we have created the following four demo applications:

1. A HTML5 video streaming application
2. A YouTube video streaming application
3. A WebRTC videochat application
4. A Chrome browsing extension

By having the WebMP working with all these, arguably different, applications, we show that it is possible to create a generic session mobility platform. Additionally, by using existing solutions, such as YouTube's player API and Google Chrome's API, we prove that such a solution is usable by third-party applications.

We're also the first to create such a generic session mobility platform (in a web environment), and the first (as we know of) that utilizes WebRTC to provide session mobility functionality.

In this chapter we will first take a deeper look into WebRTC, Socket.IO and Node, and why we chose to use these technologies. Then we'll explore the WebMP and its entities, as well as the demo applications, in more detail. Finally, we will take a look at how the implementation actually works, both by showing flow diagrams, as well as screenshots from the various demo applications.

8.2 Technology

As mentioned, our proof of concept is heavily based on the WebRTC's JavaScript API as specified in (65)[2]. Upon a migration, we use WebRTC to establish a peer connection (RTCPeerConnection), including a data channel (RTCDataChannel), between the two devices involved in the migration session.

In order to set up such an RTCPeerConnection, however, WebRTC needs a mechanism to coordinate and exchange initial information between the devices, a process known as signaling. The signaling methods and protocols are actually *not* specified by WebRTC,

[2] Note that this is the W3C's Editor's Draft published 22 March 2013. The latest editor's draft was published 03 June 2013, during which time we were finished developing the proof of concept.

instead it is up to the app developers to choose whatever signaling/messaging protocol they prefer. This led us to choose Socket.IO, a JS-library for real-time applications with a client-side library running in the browser and a server-side library for Node (which runs JS). We chose this since we wanted a lightweight, but at the same time efficient and scalable service.

First, we'll take a deeper look at WebRTC. What it is, how it works, and why we chose it as a basis for our proof of concept.

8.2.1 WebRTC

In 3.3 we talked about HTML5 and how it looks like it will become one of the dominant among applications in the future. This, and the fact that HTML5 makes it easy to deploy cross-device (mobile and PC) applications, made HTML5, and thus the web environment a good choice for our proof of concept.

Due to its dynamic and asynchronous nature, JS pointed itself out as the natural basis for our proof of concept implementation. Luckily, there's a lot of JS-libraries available, and we were happy to see that WebRTC, a JS-API still being drafted, looked promising. With its Peer-to-Peer API, enabling browser-to-browser data channel communication, WebRTC pointed itself out as a natural choice for our Peer Connection module. It was also a great motivation to take part in a 'bleeding edge technology'.

WebRTC overview

WebRTC is a free, open (ongoing) project supported by Google, Mozilla and Opera, with the following mission: *"To enable rich, high quality, RTC applications to be developed in the browser via simple Javascript APIs and HTML5."* (62)

According to HTML5 Rocks: "*The APIs and standards of WebRTC can democratize and decentralize tools for content creation and communication—for telephony, gaming, video production, music making, news gathering and many other applications. Technology doesn't get much more disruptive than this.*" (66).

The WebRTC APIs

WebRTC implements the following three APIs:

1. *MediaStream*, used to access and create a local media stream, such as from the device's camera and microphone.
2. *RTCPeerConnection*, used to establish a direct (browser-to-browser) connection between peers
3. *RTCDataChannel*, used to send (arbitrary) data between the peers in an RTCPeerConnection

The *MediaStream API* (a.k.a. GetUserMedia), includes methods for an application to take control of media input devices through the operative system in order to record the input and create a media stream. It also includes methods to convert that stream to an **object URL** (in order to make it accessible and pass it to a *<video/>* HTML element).

The *RTCPeerConnection* is the WebRTC component that handles stable and efficient communication of streaming data between peers. It is used for peer discovery, session initiation and session establishment between two peers. Further, it can be used to connect a stream from the MediaStream API and to negotiate parameters such as codec control, encryption and bandwidth management for the connection between the peers.

The *RTCDataChannel* allows for peer to peer exchange of arbitrary application data with low latency, high message rate/throughput, built-in security (DTLS) and congestion control, and optionally reliable or unreliable semantics. This functionality can be used for many real world use-cases such as gaming, real-time texts and file transfer, complementing and leveraging the RTCPeerConnection setup, which focuses on creating the connection. The RTCDataChannel API's methods are deliberately similar to WebSocket (which we'll look into later), but works a lot faster since the communication occurs directly between the browsers.

Signaling

In order to establish a direct browser-to-browser communication (the RTCPeerConnection), with, optionally, a direct data channel (RTCDataChannel), one needs a *signaling* protocol to help initialize, coordinate, control and establish this connection. See Figure 23.

WebRTC is *protocol agnostic*, i.e. this signaling protocol is *not* specified by WebRTC. Instead, it is up to the app developer to choose whatever signaling protocol they prefer, e.g. SIP, XMPP, WebSocket etc. Here, we chose a Socket.IO/Node-based signaling setup, which we'll get back to later in this chapter.

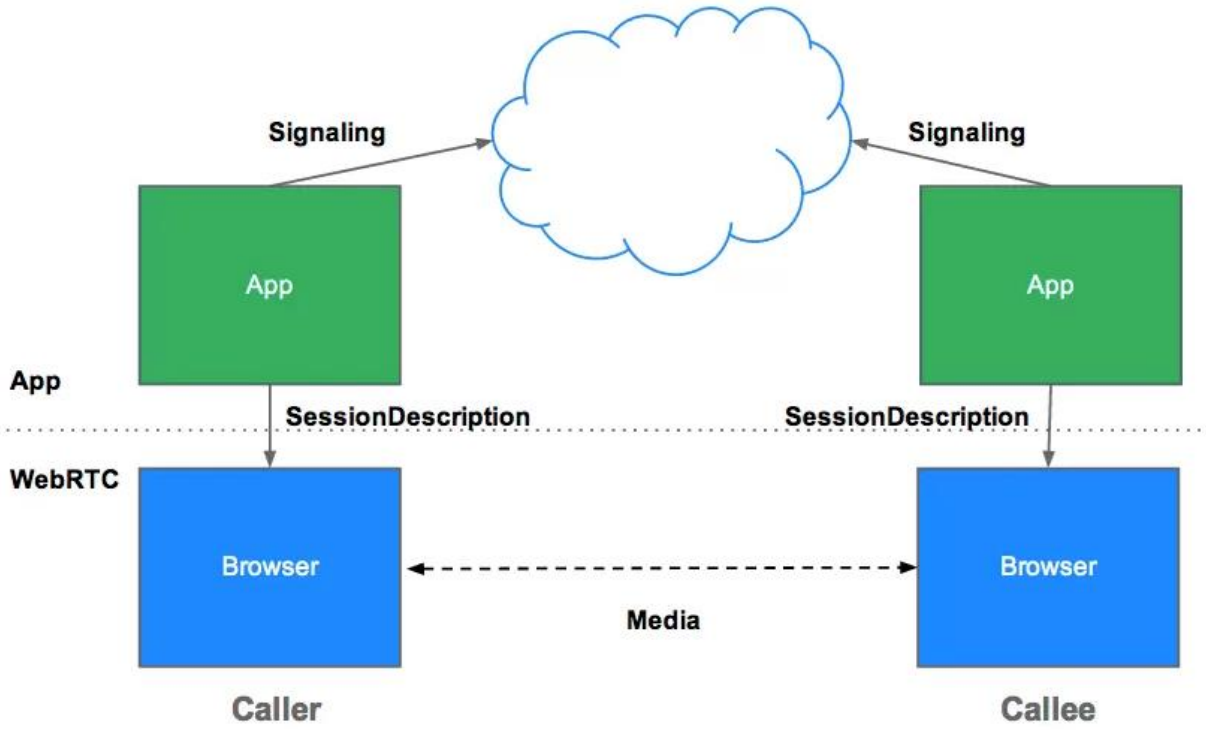

Figure 23. WebRTC architecture. Taken from (66).

Peer connection establishment

WebRTC follows an Offer/Answer model when establishing peer connections. In principle it works as follows (see Figure 24):

1. The initiating peers sends an “offer”-message to the remote peer (via the signaling server), including its supported configurations/capabilities), transport addresses and other related metadata for the session.
2. The remote peer receives the offer and creates and sends an “answer” (via the signaling server) containing the supported configurations that is compatible with the parameters supplied in the offer, as well as its transport addresses and other related metadata.
3. The initiating peer receives the answer, and a direct RTCPeerConnection is established between the peers. The rest of the session data is now sent directly between the peers.

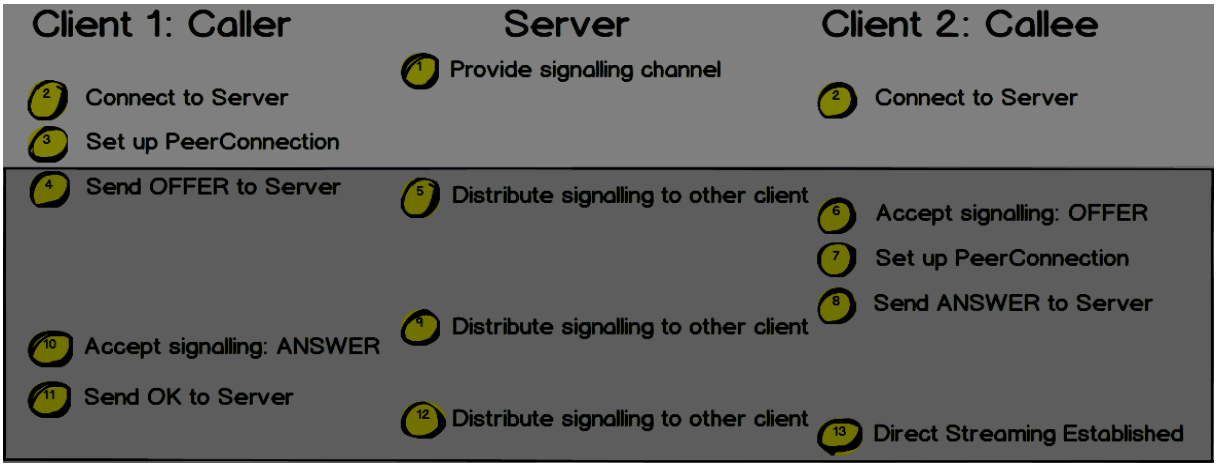

Figure 24. Example of an RTCPeerConnection session establishment with a MediaStream. Taken from (67).

This Offer/Answer model is called JavaScript Session Establishment Protocol (JSEP). JSEP allows for full control of the signaling state machine from JS. As we can see from Figure 24, this mechanism effectively removes the browser from the core signaling flow, and the only interface needed is a way for the application to pass in the local and remote session descriptions negotiated by whatever signaling mechanism is used. (68).

JSEP specifies a generic protocol needed to generate an Offer/Answer model based on the Session Description Protocol (SDP). SDP (RFC 4566) is a format for describing multimedia sessions/streams. "*It provides a general purpose, standard representation to describe various aspects of multimedia session such as media capabilities,transport addresses and related metadata in a transport agnostic manner, for the purposes of session announcement,session invitation and parameter negotiation*." (69) (70).

Hence, an RTCPeerConnection is established by sending SDP-blobs containing offers and answers between the peers, via the signaling server. When creating either offers or answers, SDP blobs is generated containing the supported configurations for the session (or in the case of an answer, supported configurations that is compatible with the parameters supplied in the offer), including, if applicable, descriptions of the local MediaStreams attached to the RTCPeerConnection, the codec/RTP/RTCP options supported by the implementation, and any transport candidates/information gathered in order to establish a direct channel. "*While the SDP format is not optimal for manipulation from Javascript, it is widely accepted, and frequently updated with new features*." (71).

NAT traversal

One of the main challenges that WebRTC had to face in order to provide a direct peer-to-peer connection between browsers, was the challenge of NAT traversal. The following RFC specifications were defined to solve this exact problem:

- RFC5389 – Session Traversal Utilities for NAT (STUN)

- RFC5766 – Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)
- RFC5242 - Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols

WebRTC takes advantage of all the above protocols to solve the problem of peer connectivity and NAT traversal.

STUN was a set of "tools" defined to solve the problem of NAT traversal by allowing a client to discover what kind of NAT lies between itself and the Internet (with the help of an external STUN server) and using its IP address and ports for itself. In order to communicate with another peers though, the other party has to also implement the STUN protocol and to be able to provide a public IP address (either their own or their NAT's). STUN soon was discovered to be insufficient to be a deployable solution for the problem of NAT traversal in some situations and led to the creation of other protocols, tools and mechanisms (72).

TURN is the successor and extension of the STUN protocol. STUN has a similar functionality, but several problems were noticed with specific NATs that STUN wasn't able to traverse. Consequently TURN was created to provide universally applicable NAT traversal (73).

TURN is a protocol that allows the host to control the operation of the relay and to exchange packets with its peers using the relay. TURN differs from some other relay control protocols because it allows a client to communicate with multiple peers using a single relay address.

The requirements for a typical TURN implementation include a TURN server with a public IP address which is accessible by everyone, whether a client lies behind a NAT or not. The client talks to the server through a mechanism called "transport address" which is a combination of IP and port, and is used both for the client and the server. Even though TURN offers a solution for NAT traversal, it does so in a very costly manner. It requires a server that will constantly work as an addressing server and route messages from one peer to the other.

Finally, we have the *ICE* protocol. ICE makes use of both STUN and TURN and it serves as a tool for other protocols to implement to achieve NAT traversal. ICE was originally meant to be used for UDP-based media streams and the SIP protocol, but it can also be extended to handle other transport protocols (such as TCP) as long as the session negotiation follows the Offer/Answer model. Since ICE makes use of two other protocols/mechanisms (STUN and TURN) to achieve the goal of Interactive Connectivity between two end-points, it should be referred to as a methodology rather than a pure protocol (74).

In WebRTC, the two peers in an RTCPeerConnection has an ICE agent each. These ICE agents are responsible for negotiating the connection path with the remote peer (with help from STUN/TURN servers) and create a list of potential connection points, so-called ICE candidates. These candidates actually constitutes all the various transport addresses (IP/port) to reach the peer. The goal is to establish a direct path, but if no direct path can be established, ICE uses an intermediary (relay) TURN server. See Figure 25. (You can read more about the WebRTC ICE interface, as well as look at a sequence diagram, in appendix, section 12.3.4.2)

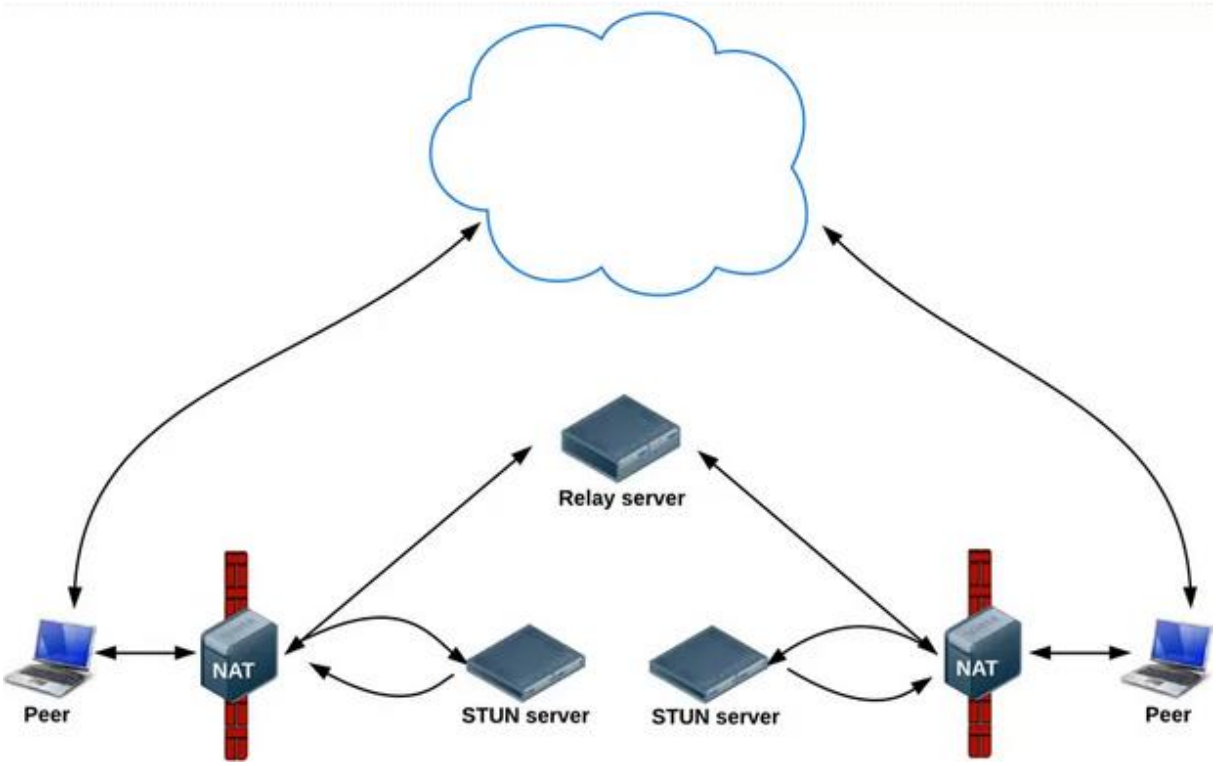

Figure 25. NAT traversal in WebRTC using ICE. Taken from (66).

WebRTC in our proof of concept

In our proof of concept, the WebMP, we use the following APIs:

- The RTCPeerConnection API. To establish a peer connection between the devices that are to participate in a migration session
- The RTCDataChannel APIs. To do the actual migration, i.e. transfer the state (and possible external view requests).

The RTPeerConnection API is going to be the basis of our solution. We are going to use the PeerConnection capabilities in conjunction with a signaling mechanism (explained in the next sub section 8.2.2) to ensure device discovery and establish an RTCDataChannel between the devices.

In order to use WebRTC's NAT traversal (ICE), and hence establish a direct data path, we are going to need a STUN and/or TURN server. We can use one of the many available free STUN servers (75), and chose to use one of Google's public STUN server. However, acquiring the resources needed for a dedicated TURN server was harder, even though we could have used an open-source server software available from Google Developers (76). However, according to Google, 92% of connection attempts performed using ICE can take place directly and without using a relay server such as TURN (77). This is why we find it acceptable and feasible to leave TURN functionality out of our proof of concept implementation, while still being able to showcase how our solution works.

Once an RTCPeerConnection has been established, thanks to the signaling server and the ICE agents, there will be an open, direct (or relayed) data channel between the devices. Now, the RTCDataChannel API will be used for the actual migration, i.e. state transfer and possible external view communication. Via this data channel, we can send any kind of data as long as it is in binary, serialized form. The technology used for serialization is not relevant or restricted by WebRTC. It depends on the underlying application and the technology it uses. For our proof of concept we are going to use JSON, which is a standard method for representing objects and serializing them, supported by JS by default (78).

WebRTC interoperability notes

For our proof of concept, we chose to limit ourselves to the Chrome environment, i.e. the proof of concept is currently only working on the PC and Android versions of Chrome. We chose Chrome because, unlike Firefox, WebRTC is available in its stable versions. Additionally, as of today, there is no complete interoperability between the different platforms. More specifically, the WebRTC DataChannel between Firefox and Chrome is not interoperable (http://peerjs.com/status). As such, there was no incentive in working on both platforms. However, it is fair to assume complete interoperability between all major browsers (and possibly other platforms as well) once the WebRTC standard is finalized (79). See appendix section 12.3.1 for more on WebRTC interoperability.

8.2.2 Node and socket.io

As mentioned, WebRTC is dependent on, but does not specify, a signaling protocol for both peer discovery and peer connection setup. Thus we were required to create such a signaling protocol. Fortunately, we had the freedom to implement it in any way we saw fit.

We want a centralized peer-to-peer architecture, as described in the solution design in chapter 7. While WebRTC takes care of the peer-to-peer part, we now have to create a globally available server that is responsible for connecting the devices, keep track of (and communicate) the associations, and aid in peer connection setup. As we are

potentially working with real-time applications with strict time constraints, the server should also be efficient (and scalable).

We ended up with the Node/Socket.IO combination, which we believe satisfied all of the above requirements, while at the same time utilized cutting edge technology. Additionally, it is implemented 100% in JS, which helped in lowering the barrier to entry, as we only had to adhere to JS. We ended up with the following signaling setup:

- Server-side: A Socket.IO-based web server running on Node (the WebMS)
- Client-side: A Socket.IO-based client/script (the WebMC)

Node

Node.js is a server-side platform built on Chrome's JS-runtime (the V8 environment (80)). It was released in 2009, and have since received a lot of attention. Node is based on an event-driven, non-blocking (or asynchronous) I/O model "*that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices*" (64).

The event-driven, non-blocking model of Node is due to the fact that Node *is* JavaScript, which is based on asynchronous functions. I.e. instead of using separate threads and waiting for I/O operations, a function is attached to the finish events, creating an event loop, where events are "fired" whenever a function is finished. Thus, Node can still make use of its processing power when the server is waiting for any other operation. This results in an optimized performance while avoiding a multithreaded overhead, and "*makes Node.js scalable to millions of concurrent connections.*" (81).

Hence, Node clearly has the scalability capabilities we need in our centralized peer-to-peer model. Our Node server (the WebMS) should be able to handle multiple concurrent connections, and be able to aid in establishing peer connections upon migrations, satisfying real-time constraints.

The next challenge was to find a way to connect the clients and the server. By leveraging the fact that Node runs on JS, we ended up with Socket.IO, a Node.JS-project based on the WebSocket API, for real-time web applications. This way, we could easily create a JS-based socket connection between clients and server.

Socket.IO

As mentioned, Socket.IO is based on the WebSocket API. *WebSocket* is developed as part as the HTML5 initiative, and is the "*next generation method of asynchronous communication from client to server.*" (82), compared to Comet and Ajax. It defines "*a full-duplex single socket connection over which messages can be sent between client and*

server“ (58), which enables unprovoked bi-directional message pushing between client and server, (unlike e.g. AJAX, which requires that a request is made by a client).

Like Node, WebSocket is event-based, which creates a very simple API for the developer. E.g. whenever a socket client receives a message, an event “onmessage” is triggered, and the script does whatever the developer has specified.

Socket.IO builds on the WebSocket API, but adds additional features that are not provided by the WebSocket API out of the box, e.g.:

- *Fallback transports*. In order to provide real-time connectivity to every browser (PC or mobile), Socket.IO selects the most capable transport mechanism at runtime. Should the client not support WebSocket, Socket.IO can select another transport protocol (e.g. Flash Socket, AJAX long polling, JSONP polling etc.). Thus, Socket.IO is “*blurring the differences between the different transport mechanisms.*” (63).
- *Heartbeats, timeouts and disconnection supports*. Socket.IO helps specify values and configurations for keeping track of a connection, assuring that a client is alive and active, by having it send so-called heartbeats at predetermined intervals.

(83) (84)

Socket.IO has two parts; a client-side JS-library and a server-side JS-library, with APIs that are “*nearly identical*” (85). This was a great advantage to us when using Node, as we could create *nearly identical* client- and server-side code, assisting in a faster deployment.

Node and Socket.IO in our proof of concept

To summarize, Node is used to run a scalable, efficient, event-based Socket.IO Migration Server (WebMS). On client-side, a Socket.IO-based script (WebMC’s Signaling Module) is used to connect to the WebMS.

This socket connection enables a device to advertise itself and discover other devices. In order to keep connected, the client needs to send heartbeats to the server. Whenever a new connection or disconnection (e.g. a timeout) event occurs, the WebMS is responsible for multicasting this update to the relevant devices, ensuring an up-to-date association list at every client at any time.

In addition to enabling device discovery, the WebMS will assist in initial RTCPeerConnection establishment, by acting as the signaling server. I.e. it will forward the initial Offer/Answer messages between the devices that are to participate in a migration session. When the peer connection has been established, the devices communicate directly with each other, i.e. the not via the WebMS anymore. In addition to speeding up the peer-to-peer communication, this also relieves the WebMS for additional workload (state transfer etc.).

8.3 Setup

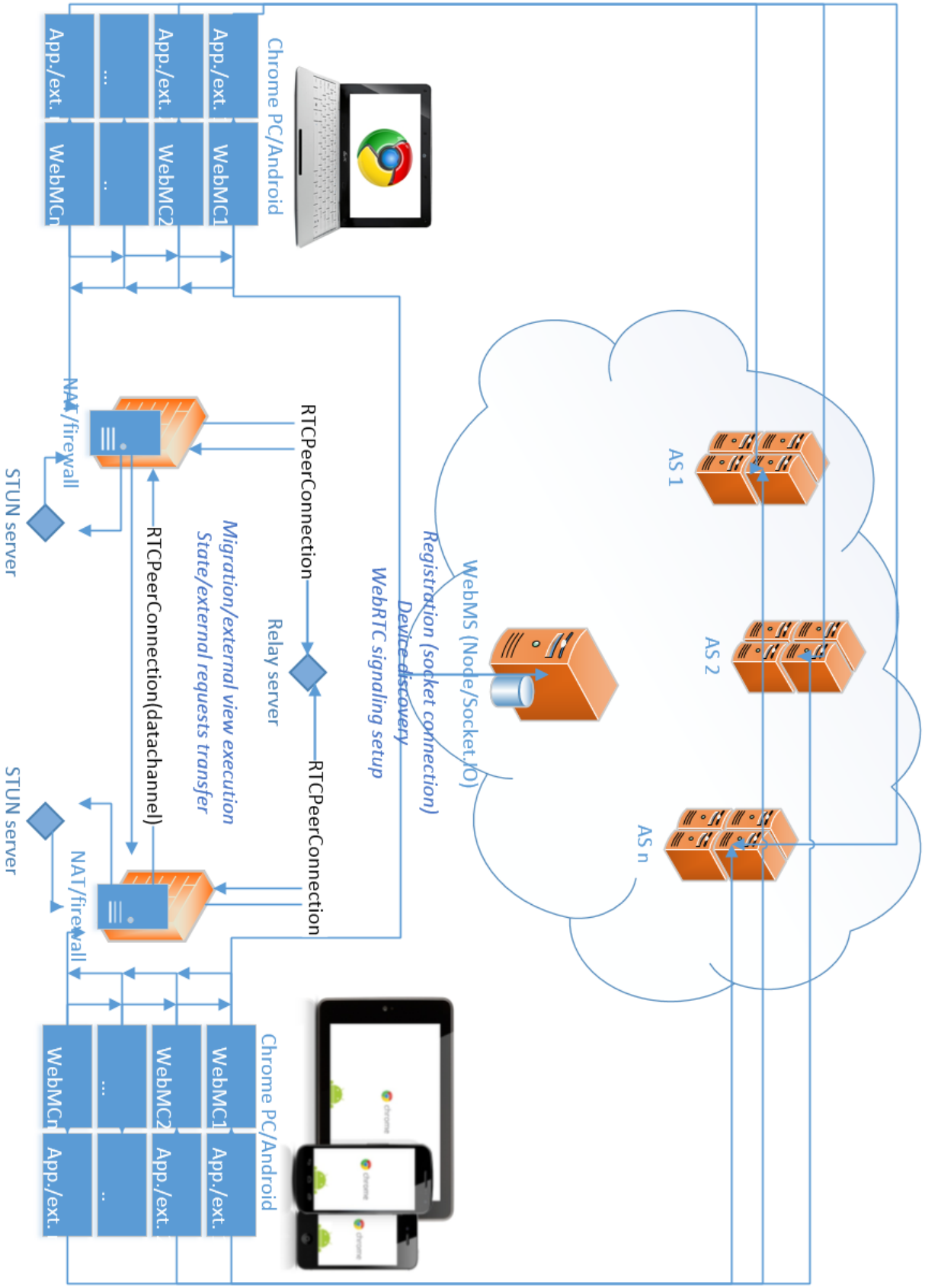

Figure 26. The Proof of Concept Architecture, showing a Web Migration Platform realized with WebRTC and Node/Socket.IO.

Figure 26 shows the architecture of our implemented proof of concept, the WebMP. It realizes cross-device, cross-domain session mobility for web applications (in the web environment) and consists of the WebMC clients and the WebMS server. The server, as well as the demo applications, are hosted by Telenor. For more about the deployment, see appendix, section 12.3.2.

8.3.1 The WebMS

The Web Migration Server (WebMS) has similar responsibilities as the MS described in the previous chapter. It is purely built in JS, and runs in the Node.js platform. The WebMS uses Socket.IO to establish and control socket connections with the application clients/devices. It is responsible for:

- Registering user's devices/applications to the migration server
- Keeping the connected devices/applications updated about the other connections
- Serve as a WebRTC signaling server for setting up WebRTC peer connections between the clients, used for session transfer and/or external view session establishment

We refer to appendix, section 12.3.3.1 for additional WebMS code documentation.

8.3.2 The WebMC

The Web Migration Client (WebMC) has similar responsibilities as the MC described in the previous chapter. However, unlike the MC, we do not implement any native module. This requires that the application must be running on the target device. For our proof of concept, we argue that the implementation of the actual session mobility/migration features is of more importance, thus we focused solely on this. If one should continue working with our proof of concept, adding a native module can be appropriate further work.

The WebMC is implemented as a JS-library consisting of two scripts (see Figure 27):

- *Migration_signaling.js.* Connects to and communicates with the WebMS by using Socket.IO. This enables the application to advertise itself and discover other devices.
- *Migration_peerconnection.js*. Creates the RTCPeerConnection and the RTCDataChannel between the devices upon a migration.

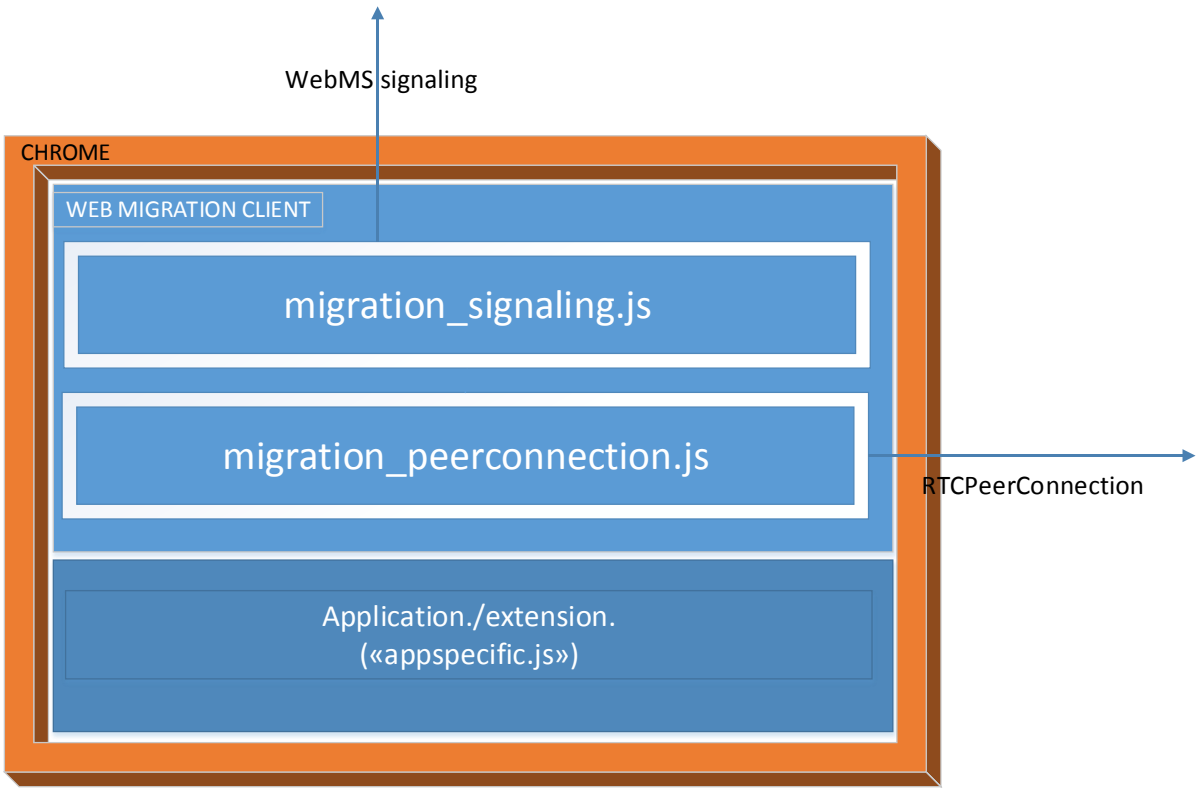

Figure 27. The WebMC and its two scripts. These scripts acts as a migration extension to a web application.

The two scripts together constitute a "migration API", with methods the application developer can call in order to migrate the application from one device to another. The Application-Interface Module now becomes this interaction, as well as the application-specific JS-methods that we require the application developer to implement.

We refer to appendix, section 12.3.3.2 for additional WebMC code documentation.

8.3.3 The demo applications

In order to showcase our implementation, we implemented some demo applications that were to include the above-mentioned client-side scripts and implement some application-specific methods to make it work. All these demo applications includes and uses the *exact same* scripts (migration_signaling.js and migration_peerconnection.js), showcasing the generic property of our implementation. What makes it work is the implementation of the required application-specific part, such as state retrieval and stateful launch. Se appendix, sections 12.3.3.3 and 12.4 for code documentation of the application-specific script, and a "how-to-demo" tutorial, respectively.

We developed the following demo applications:

- A HTML5 video streaming application
- A YouTube-video streaming application
- A WebRTC-videochat application
- A Chrome browsing-extension

HTML5 video streaming

A simple HTML5 video streaming application, using HTML5's <video/> element (86), i.e. not requiring any plugin for playing a video. We provide a list of videos the player can play, but you can play any HTML5-supported video by providing an URL.

We implemented the following migration features on this demo application:

- Full migration. The ability to transfer an ongoing HTML5 video playback from one device to another, in a paused or immediate mode.
    - I.e. we create a state object containing the URL of the currently playing video, as well as the current time and the current volume of the playback. This is read on the receiving side and results in the target device continuing the video playback from the exact same position as it was left off at the source device.
- External view. The ability to transfer an ongoing video playback (paused or immediate) to a target device, while controlling the playback from the source device. At any time, the user can either finish the migration to the target device, or exit it, retrieving the view on his/her source device.

YouTube video streaming

The YouTube player is very similar to the HTML5 video streaming application. Only here, we replace the HTML video element with an embedded YouTube-player using SWFObject (87), a JavaScript Flash Player and embed script, and access and manipulate the player via YouTube's JavaScript API (88).

WebRTC-videochat

In the videochat demo application, we have created a simple videochat application using WebRTC (thus also utilizing WebRTC's MediaStream API).

We implemented the full migration-feature in the videochat application. This enables a user to migrate an ongoing videochat session from one device to another. As previously mentioned, we have suggested that RTC-apps implement an additional migration handling mechanism in order to make the migration seamless from both point of view. Our videochat application migration works in the following way:

1. Assume two clients/devices, A1 and B1 (user A and user B), are in a videochat session using our videochat application. User A now wants to migrate to device A2.
2. Before migrating, the videochat application sends a notification (via the videochat signaling server) from A1 to B1 to notify about the migration. Hence, B1 is made aware of the migration. It communicates this to user B, and waits for a new call from user A (now on device A2).
3. Simultaneously, device A1 and A2 uses the WebMC to establish a peer connection and transfer the state (i.e. the URI of B1, as well as the private chat log history). A2 then launches, i.e. displays the private chat log and calls B1.
4. The session is continued between user A and B on devices A2 and B1, respectively, keeping the users well-informed throughout the migration.

Chrome browsing extension

We also implemented a Chrome browsing extension (89), using Chrome's API.

As with the videochat, we only implemented the full migration-feature in the browsing extension. This enables a user to migrate an ongoing browsing session from one device to another. More specifically, the browsing state we made is an object containing of all the open tabs, as well as all the cookies and the history elements. When received, the target device continues the browsing session by opening all the tabs, and storing the cookies and history elements.

During implementation of this extension, we stumbled upon size and bandwidth limitations in the RTCDataChannel. This resulted in us having to send the state over several packets, in a controlled speed (to avoid exceeding the bandwidth limitations). See appendix, section 12.3.3.4.

8.4 Execution

In this section, we will present both diagrams and screenshots showcasing how our proof of concept works. We will clarify where our proof of concept differs from the proposed solution design mentioned in the previous chapter, and why. Implementation-specific parts will also be explained.

8.4.1 Connection/disconnection

In our proof of concept, there is no single, common migration client for every application. Instead, every web application is required to include the WebMC JS-library, as well as implement some required, application-specific methods.

When a user starts a migration-enabled web application (i.e. loads the page or extension), he/she should be prompted to provide his/her "migration user" credentials in order to log in to the WebMS, and thus carry out the various migration functions offered. See Figure 28 and Figure 29 for examples of how this is done in our YouTube-video demo application. This will result in the WebMC connecting to the WebMS, advertising the device, as well as continuously updating the application with the available devices.

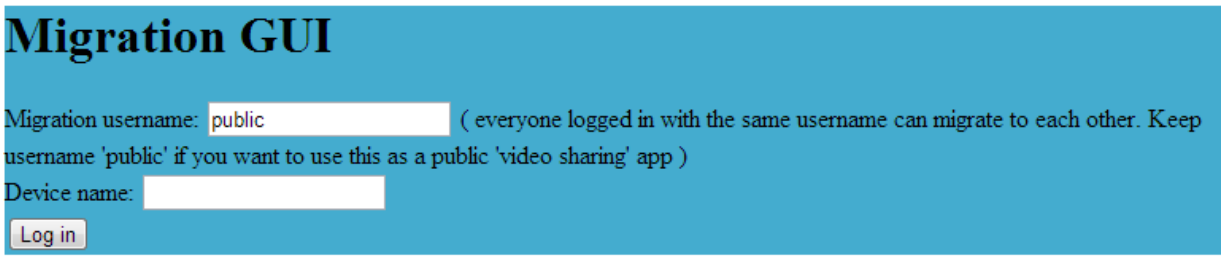

Figure 28. The "Migration GUI" in our Youtube-video demo application. The user simply is displayed a login form in order to connect to the WebMS.

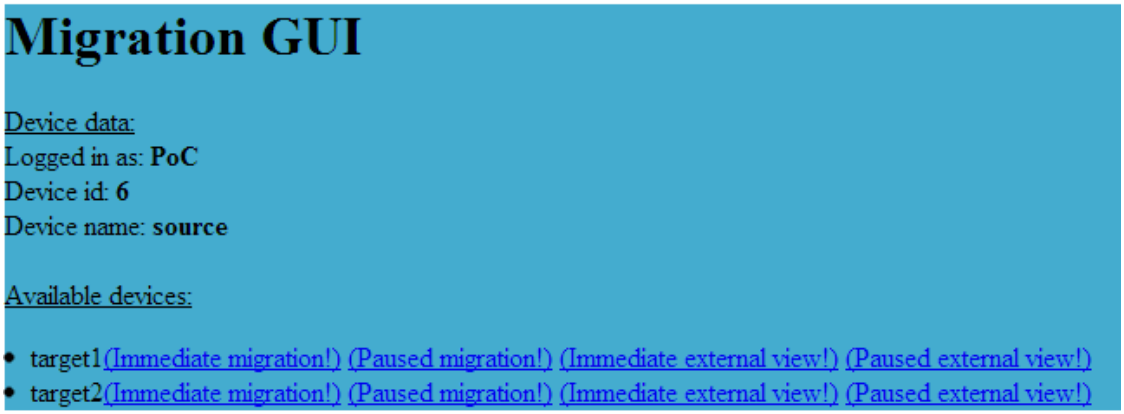

Figure 29. The "Migration GUI" after a user has logged in.

In Figure 29 we see that the "Migration GUI" displays its connection information and a list over available devices to which the user can migrate to. Here, the user is given many alternative migration alternatives.

WebMC connect/disconnect

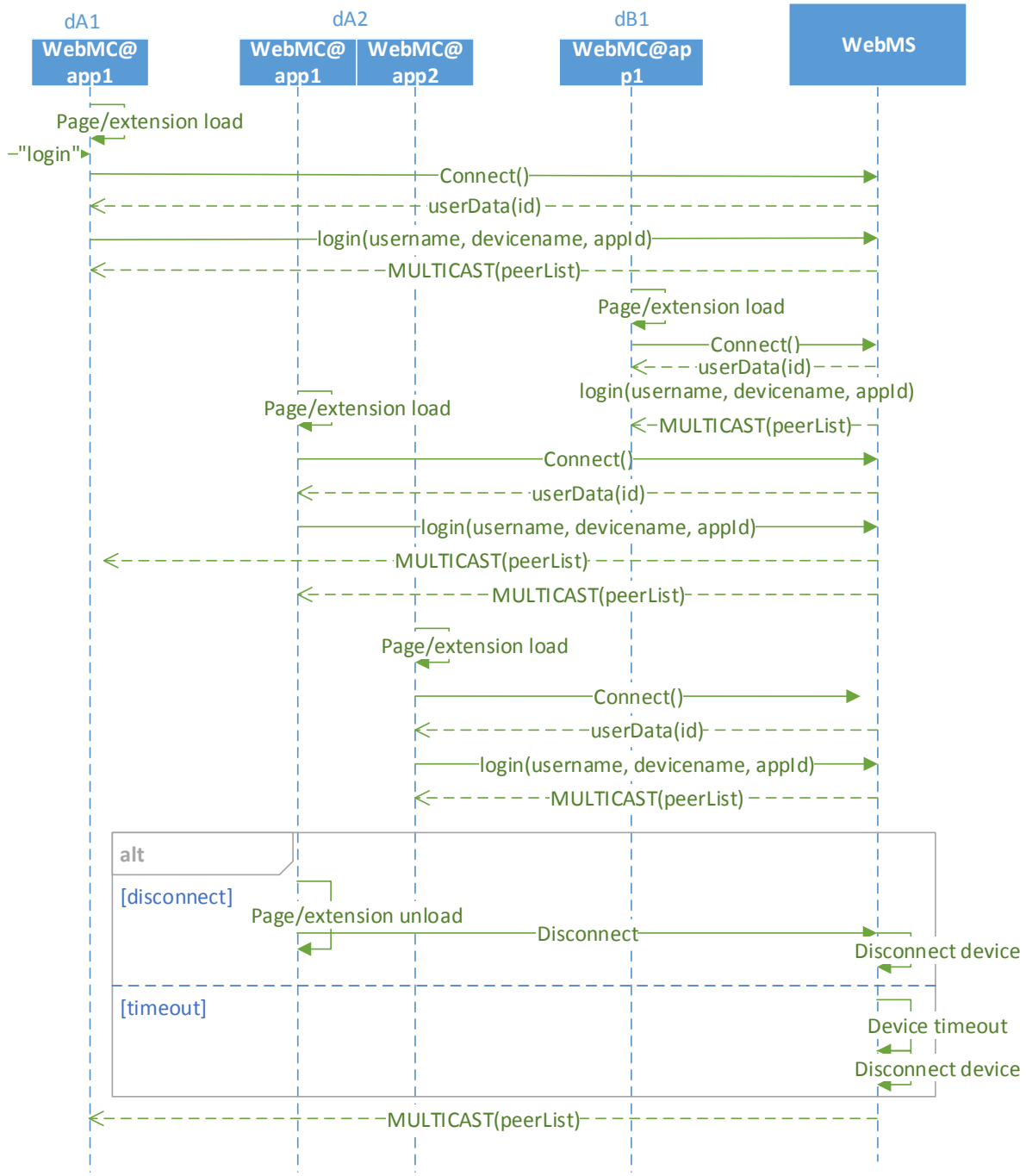

Figure 30. Device connection/disconnection & discovery in our proof of concept

Figure 30 shows how devices/applications connect to the WebMS. In the context of the web environment, since each web application/extension includes the generic WebMC script, every open tab or extension constitute a single socket connection with the WebMS.

Devices dA1 and dA2 belong to user A, while device dB1 belongs to user B. Each device may have several active applications, shown in parenthesis (in this scenario dA1 has two; app1 and app2). Assume that there's initially no other devices connected to the server.

In the diagram, we see that the user provides his/her credentials and logs in. This triggers a WebMC-initiated Socket.IO connect(), which the WebMS responds to with the socket connection's id. This id is later used when initiating migrations (RTCPeerConnections) between devices. Socket.IO connection events gets fired as soon as the TCP connection is established, i.e. there is no way to send additional data when connecting. This is why the WebMC emits a "login"-after the socket is established, which is the *actual* login. This login-message passes the user- and device name, as well as the application id to the WebMS, which stores the information and multicasts the new connection to its relevant, associated connections. Upon a WebMC socket disconnection, the WebMS removes the connection and notifies its relevant associations. As such, other than the initial connection, the WebMP connection/disconnection & device discovery works similar to the proposed solution design (Figure 16).

WebMC heartbeat/timeout

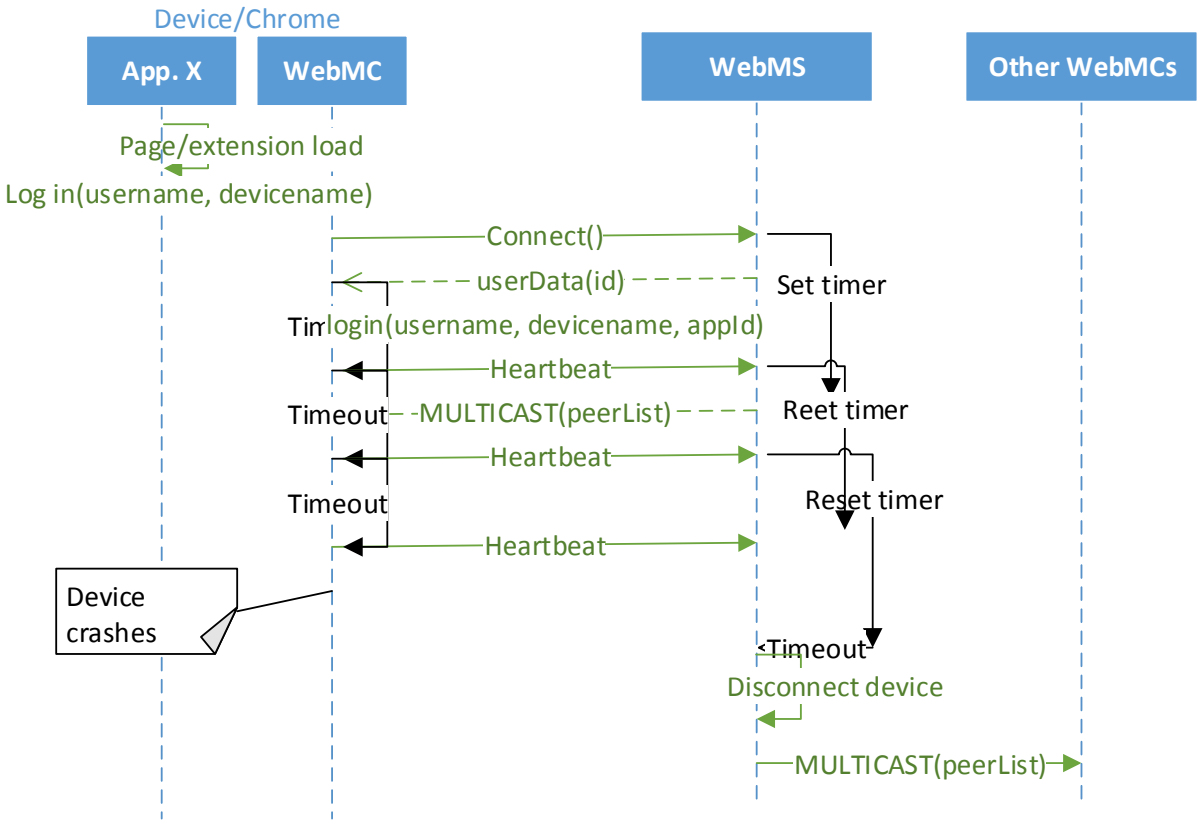

Figure 31. Heartbeat and timeout functionality in our proof of concept

Figure 31 shows the heartbeat and timeout functionality in the WebMP, which uses built-in Socket.IO-functionality (the heartbeat/timeout functionality in the solution design is actually inspired by this Socket.IO-functionality). Here, we used default Socket.IO configuration values (90), which sets *heartbeats* to true by default. Additionally, Socket.IO sets *reconnect* to true, which means the Socket.IO client will automatically reconnect should it detect a dropped connection or timeout.

8.4.2 Migration

In the WebMP, the WebMC, with initial setup help from the WebMS, establishes an RTCPeerConnection with an RTCDataChannel. The actual migration session happens over this data channel.

As in the solution design chapter, we will look at two migration scenarios; a full migration and an external view migration. We also refer to appendix, section 12.3.4.1, for

the WebMC state machine, specifying the states, and the events that triggers state change during a migration. In addition to showing call flows, we will also display screenshots from our demo applications.

Full migration

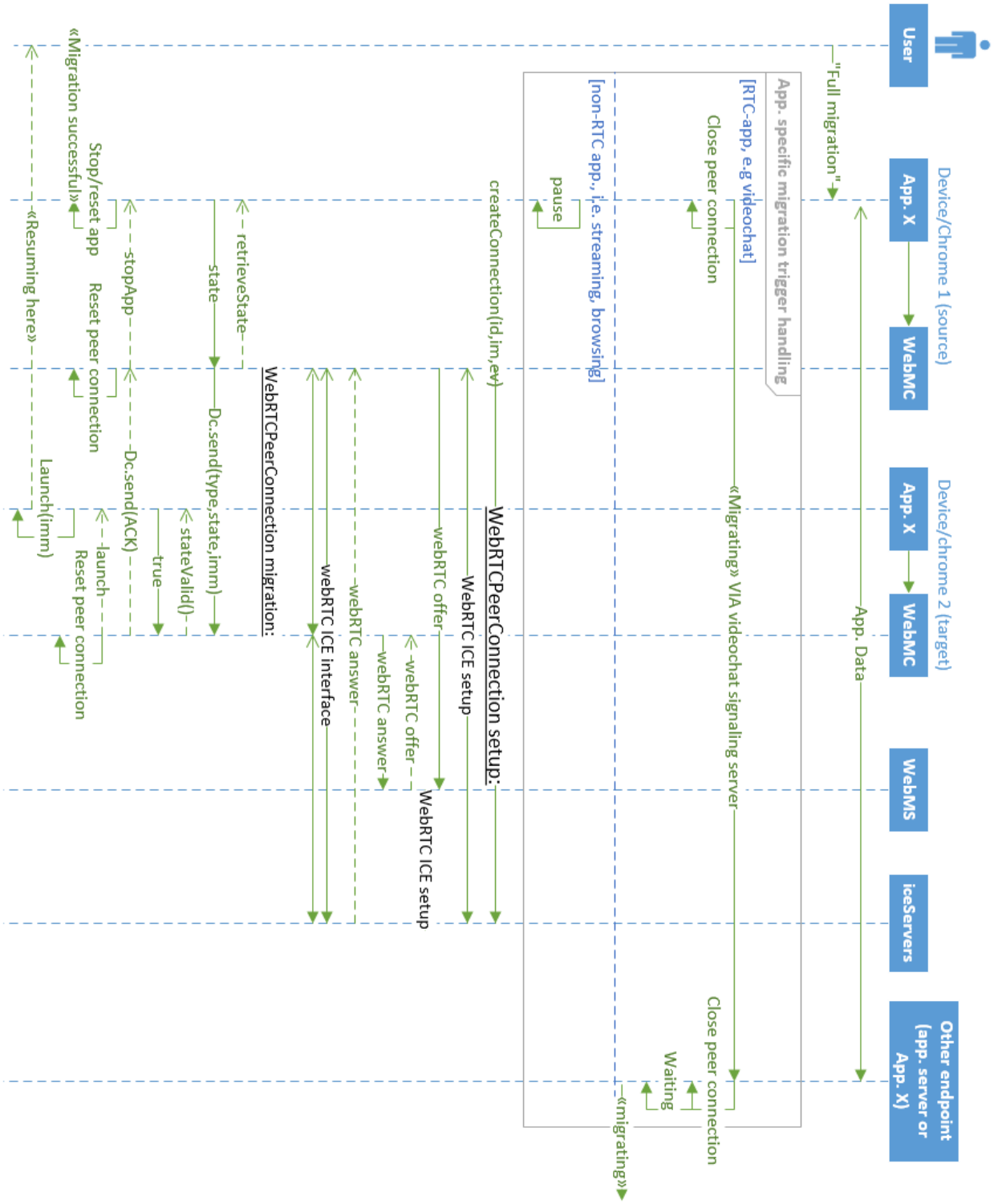

Figure 32. Simplified sequence diagram for a full migration in our proof of concept.

Figure 32 shows a simplified sequence diagram where a user fully migrates “App. X”’s session from source to target device. The diagram omits the details of the WebRTCPeerConnection setup, as well as certain WebRTC-specific details that happens during the actual migration. We refer to appendix, section 12.3.4.2 for more detailed sequence diagrams.

Assume the two WebMCs are connected to the WebMS, and that both are registered to the same user. When the user requests a migration to the target device, “App. X” is in an ongoing application session. We divide between RTC-apps (i.e. the videochat app) and non-RTC-apps (i.e. the video streaming apps and the browsing extension). If “App. X” is one of the non-RTC-apps, it is simply paused before migration, if applicable. Else, we are going to migrate a videochat session. In our videochat demo application, we send a “migrating” message (via the videochat server) to the remote videochat peer, notifying about the upcoming migration. This triggers the remote peer to enter a “wait” state, displaying a message to the remote user about the ongoing migration (see screenshots later in this section).

A “createConnection”-method is then called from the appspecific script, specifying the target device’s id, and the type of migration (i.e. Boolean parameters corresponding to immediate and external view). The createConnection method sets up the RTCPeerConnection and RTCDataChannel, using the corresponding WebRTC APIs. Here, the source sends an Offer to the target via the WebMS. The target receives this offer, creates and responds with an Answer-message (also via the WebMS). These messages contain SDP-blobs, specifying the type of offer/answer, the capabilities of the device for data channel establishment, as well as the ICE candidates that the ICE agents (working in the background) has gathered.

Once the source WebMC receives the Answer, the RTCPeerConnection can been established (and the RTCDataChannel can be opened), and the state transfer can start. The WebMC calls the application-specific retrieveState() method, which gathers and returns the state of “App. X”’s session. This state is serialized (using JSON) and sent over the data channel. Upon reception of this state, the target WebMC passes the state to the application, which checks and sets the state. If the state is OK, the target replies with an “ACK” over the data channel, before it tells “App. X” to launch and resets the RTCPeerConnection. Simultaneously, the source WebMC tells “App. X” to stop (reset) and resets its RTCPeerConnection upon reception of the “ACK”.

Videochat migration

FIGURES 1, 2 and 3 show how a migration in the videochat demo application is done.

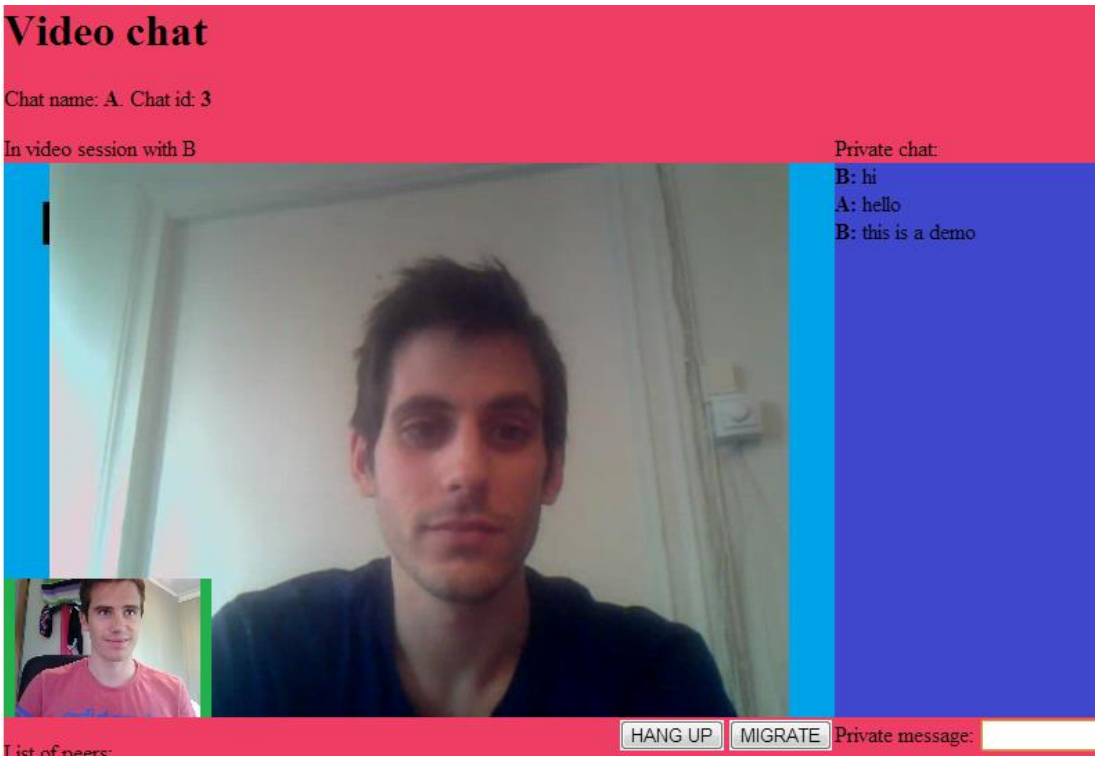

Figure 33. Videochat demo. User A and B in a videochat (and private chat) session.

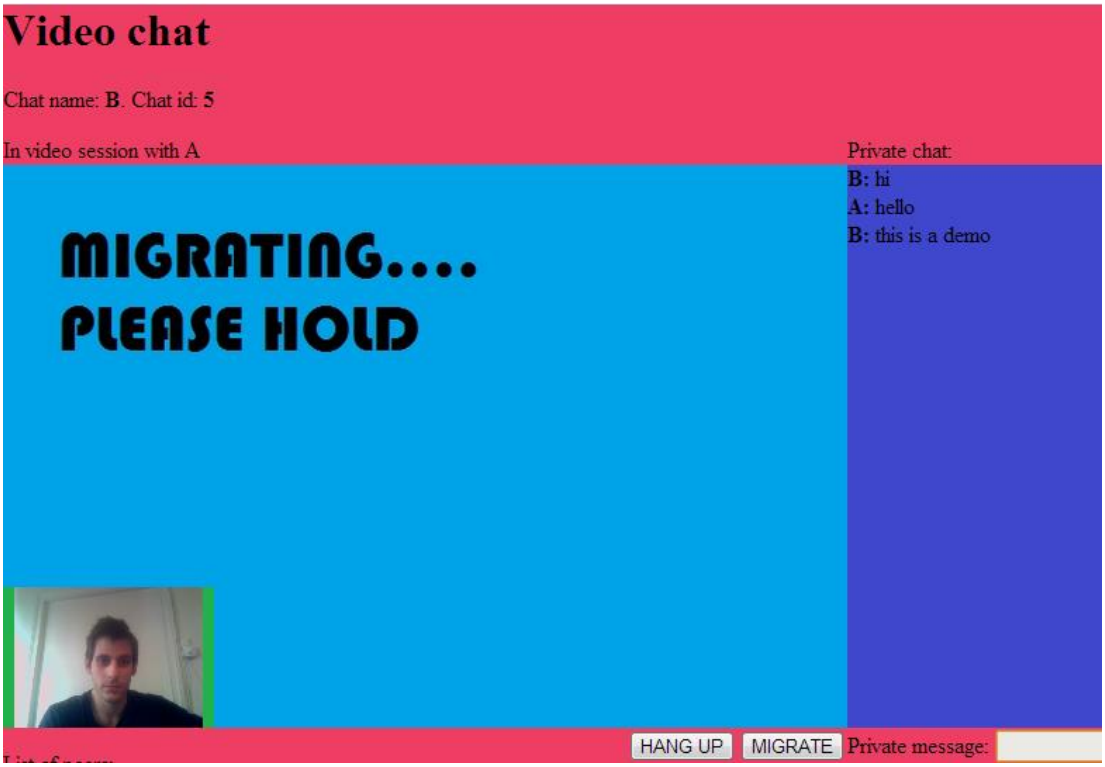

Figure 34. Videochat demo. Videochat app at B displaying a message about user A's migration

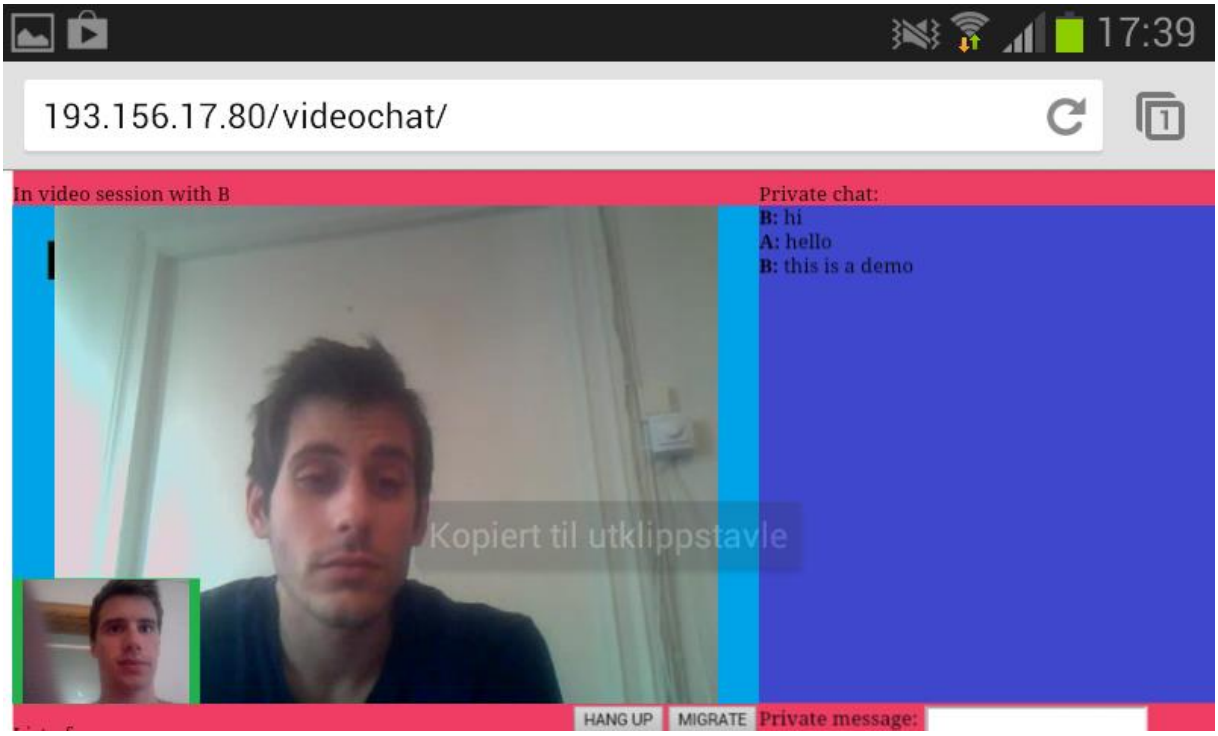

Figure 35. Videochat demo. User A has resumed the session from his mobile device, keeping the private chat from the previous device.

In Figure 33, user A and B are in a (desktop/desktop) videochat session (with a private chat/IM). Now, user A is on the move, and wants to migrate to his mobile device to continue the videochat from there.

Upon triggering the migration, a message is sent from A's videochat app to B's videochat app, notifying about the upcoming migration. This triggers B's videochat app to enter a "wait" state, as well as to display a message to user B about the ongoing migration. See Figure 34.

Using the WebMP, user A's desktop device creates a peer connection with the mobile device and transfers the state. The mobile device launches upon this state by filling in the private chat log and calling User B. The videochat application at User B answers the call, and the videochat session continues. Figure 35 displays the resumed session from user A's mobile device. Here, we can see that the very same private chat log is transferred.

Browsing migration

Figure 36, Figure 37 and Figure 38 show a migration of a Chrome browsing session, using the Chrome browsing demo extension.

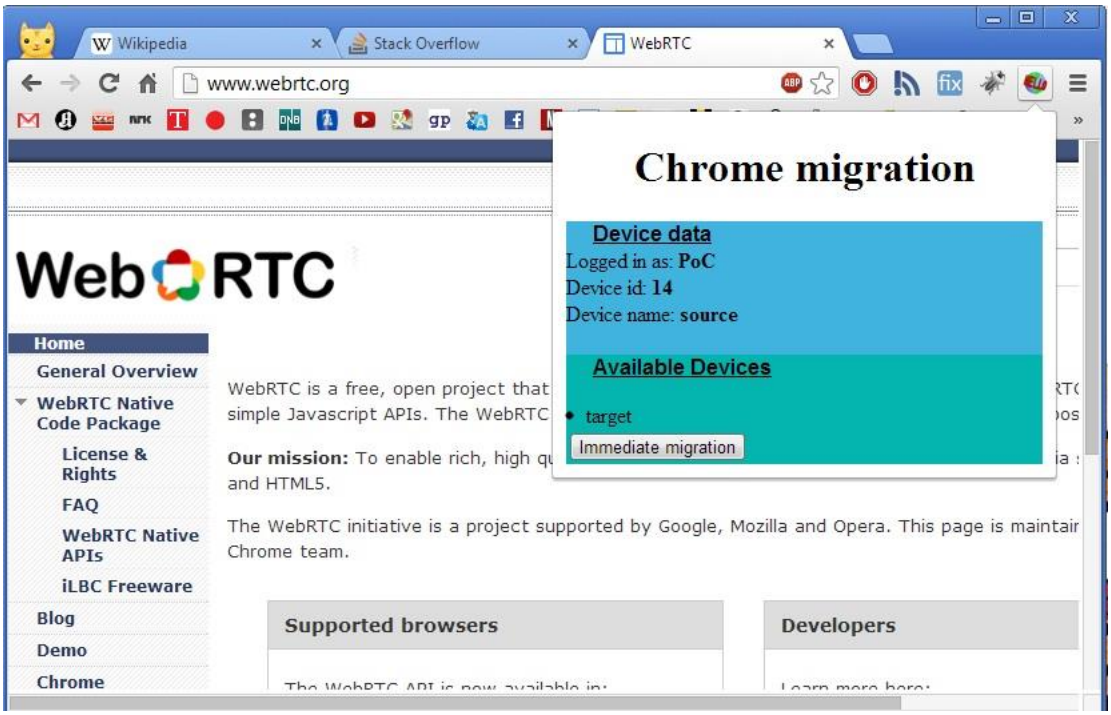

Figure 36. Browsing extension demo. User A at "source" device upon migration

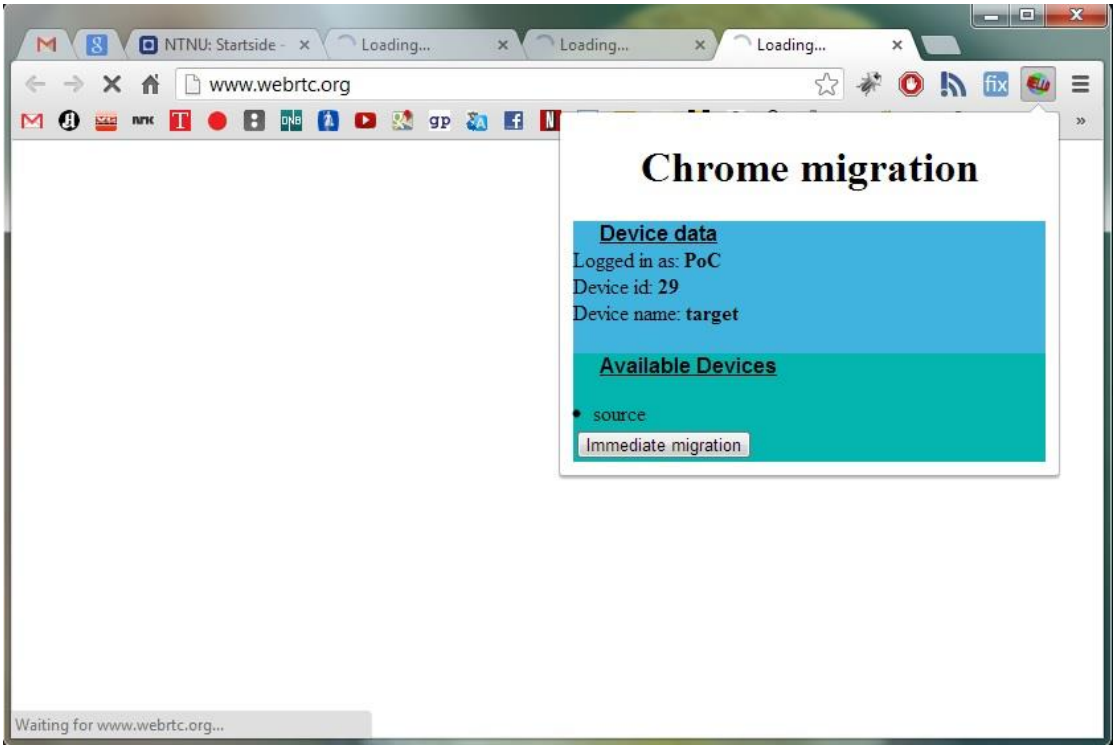

Figure 37. Browsing extension demo. Launching at "target" device.

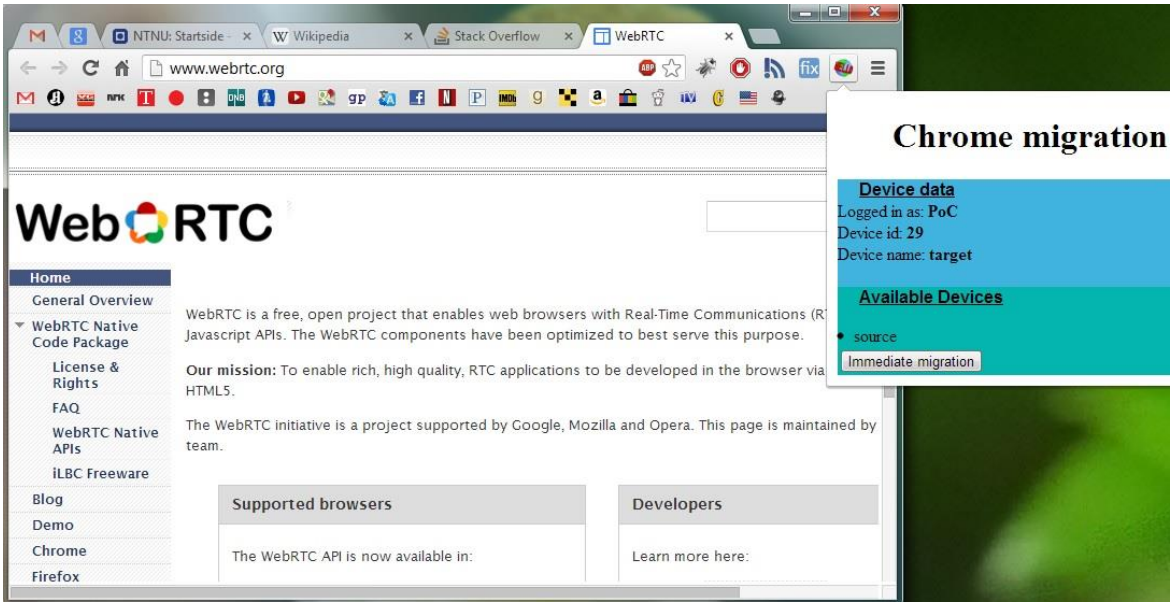

Figure 38. Browsing extension demo. The very same tabs are launched.

In Figure 36, User A is browsing on his "source" device, having three tabs open. Now, User A wants to continue this browsing session, keeping his cookies and history files, from the "source" device. He/she opens the Chrome demo extension, logs in with device name "source", and selects the "target" device. Then, by using the WebMP, the "source" device creates a peer connection with the "target" device, and transfers all the tabs, cookies and history files to the stationary device. When the stationary device have received all these files, it launches, storing the cookies and history files, as well as opening the tabs. Figure 37 is a screenshot of the moment when the tabs are opened at the "target" device, and Figure 38 shows that these in fact are the very same tabs that were open at the "source" device.

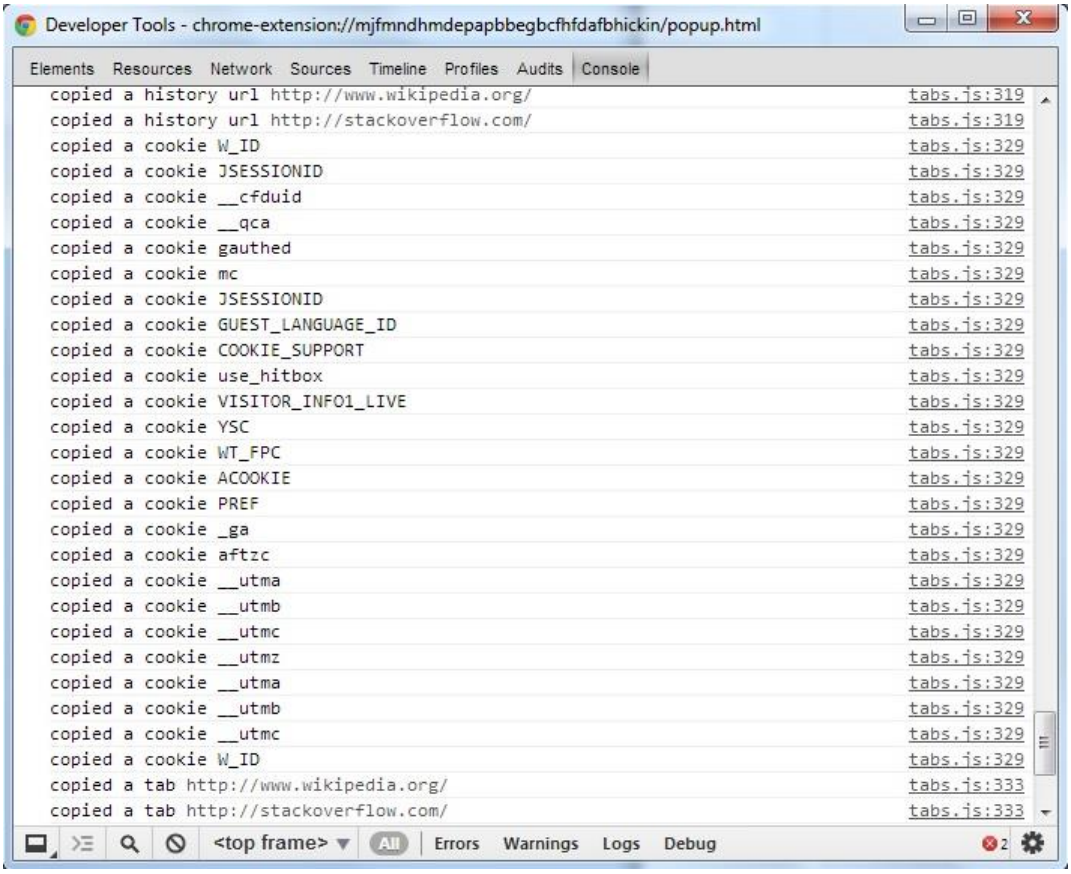

Figure 39. Browsing extension demo. Console log at "target".

What's special with the browsing migration compared to the other demo applications, is the need for sending the state over several messages, in a controlled matter. This must be done to avoid exceeding the RTCDataChannel's size and throughput limits (see appendix, section 12.3.3.4). Figure 39 shows the console log output on the receiving side. After having received and stored every history object, cookie and tab, the extension copes the history and cookie files and opens the tabs.

External view

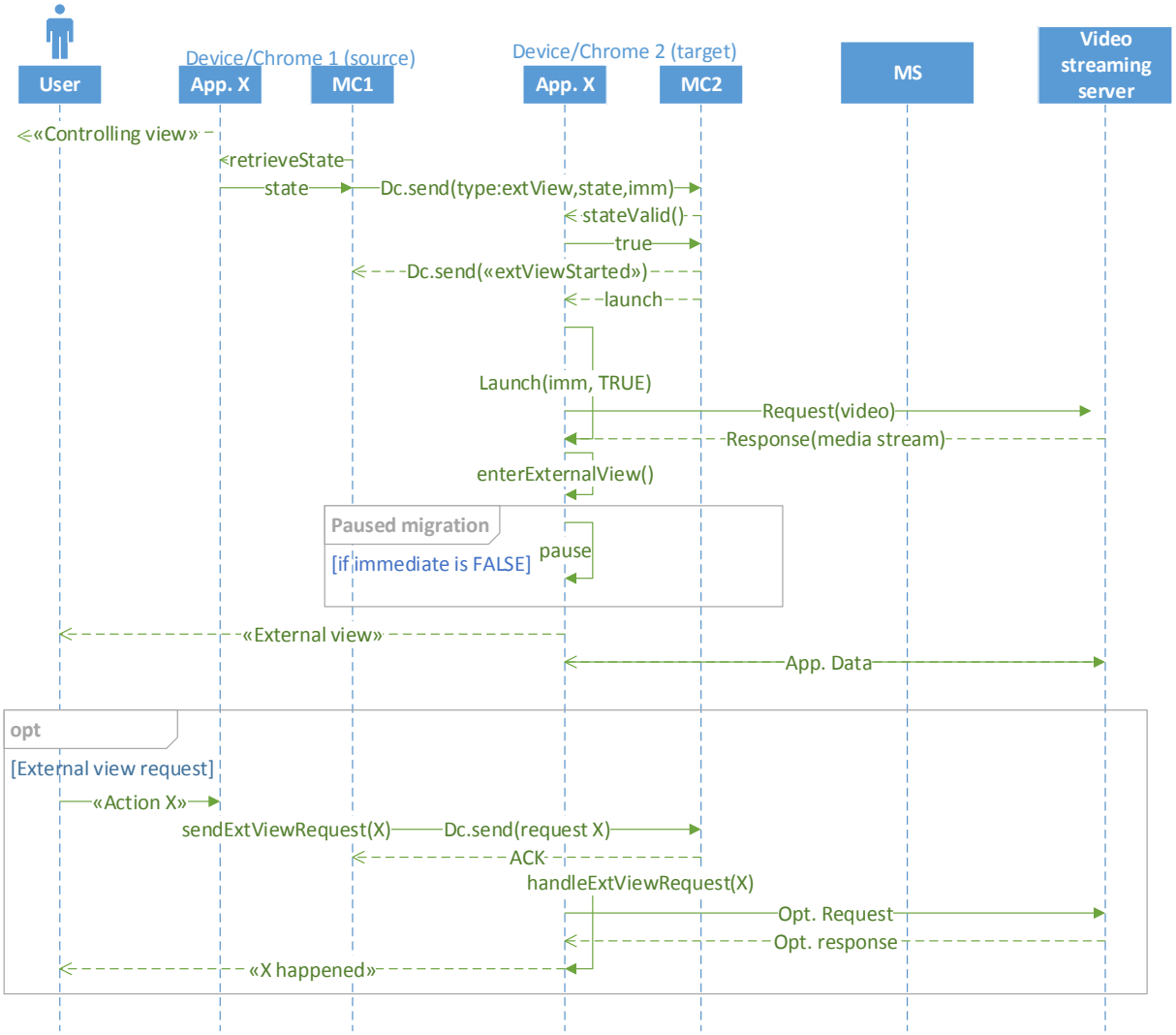

Figure 40. Proof of concept external view establishment and external view requests

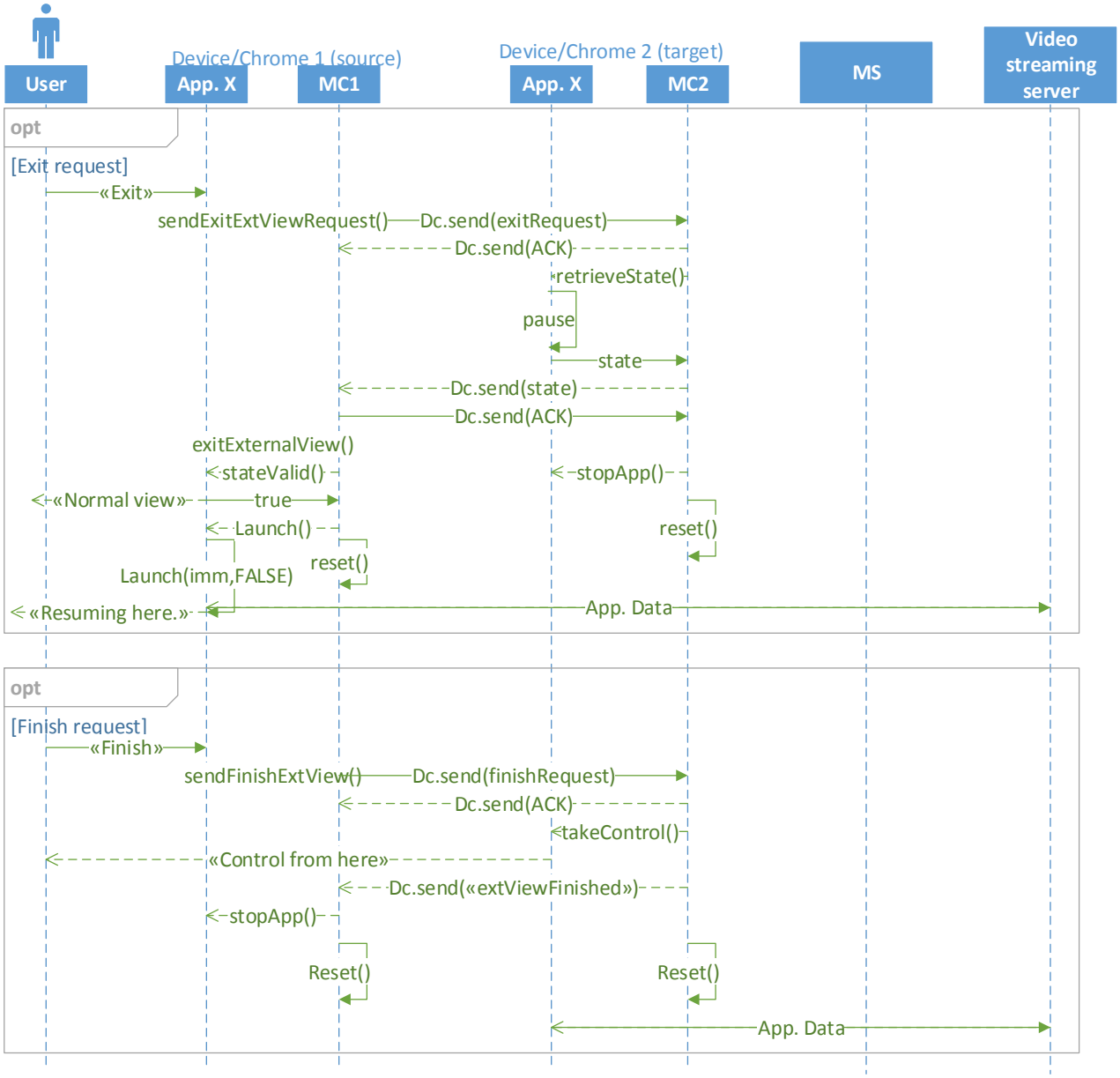

Figure 41. Proof of concept external view finish and exit flow

Figure 40 and Figure 41 show how an external view migration is done in our proof of concept, which in fact is very similar to the proposed solution design. In this case, the source device has already entered a so-called "controlling view". When the target device receives an external view migration request, it will enter an *external view* (or "controlled view") upon launching. Note, how the controlled (and controlling) view should be is entirely up to the application developer (e.g. in our video streaming demo applications, we removed all the controls and just displayed the video player). Since we're now in an external view session, the RTCPeerConnection between the source and target device is not closed, but kept open in order for the source (or controlling) device to send external

view requests. Now, the user is be able to do the following (as explained in the previous chapter):

1. Send external application (action) requests (e.g. video playback actions).
2. Exit the external view
3. Finish/complete the migration

We differentiate between these in the WebMC with JSON, giving the messages sent over the data channel different "type" attributes (in the external view case, we have the message types "request", "exit" and "finish"). Figure 40 shows how external action requests are done, while Figure 41 shows how the finish and exit flows are executed.

External view in YouTube-video player

We implemented external view in both the HTML5- and YouTube-video streaming demo applications. The figures below show an external view session in the YouTube-application.

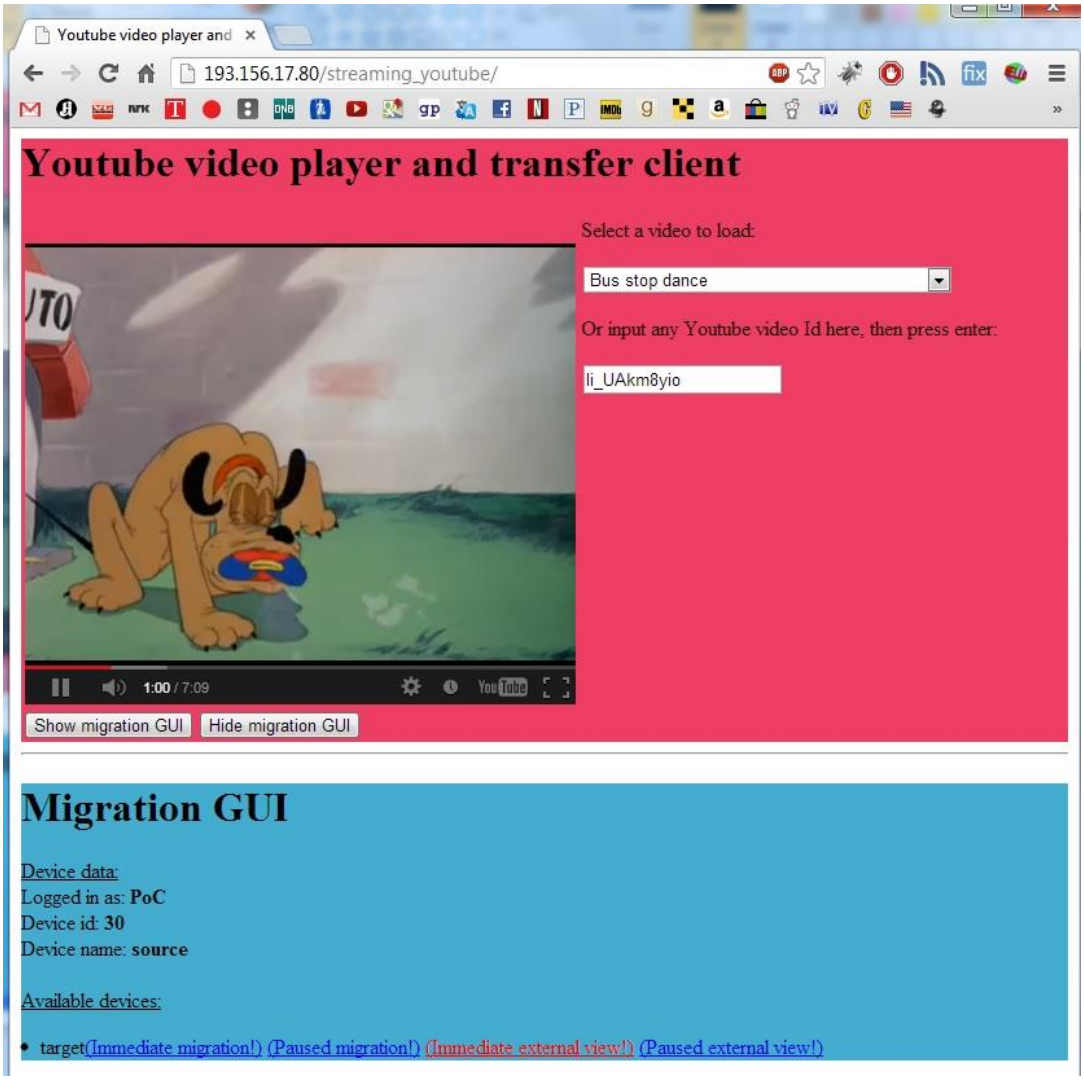

Figure 42. YouTube external view demo. User is choosing to establish an external view session with target.

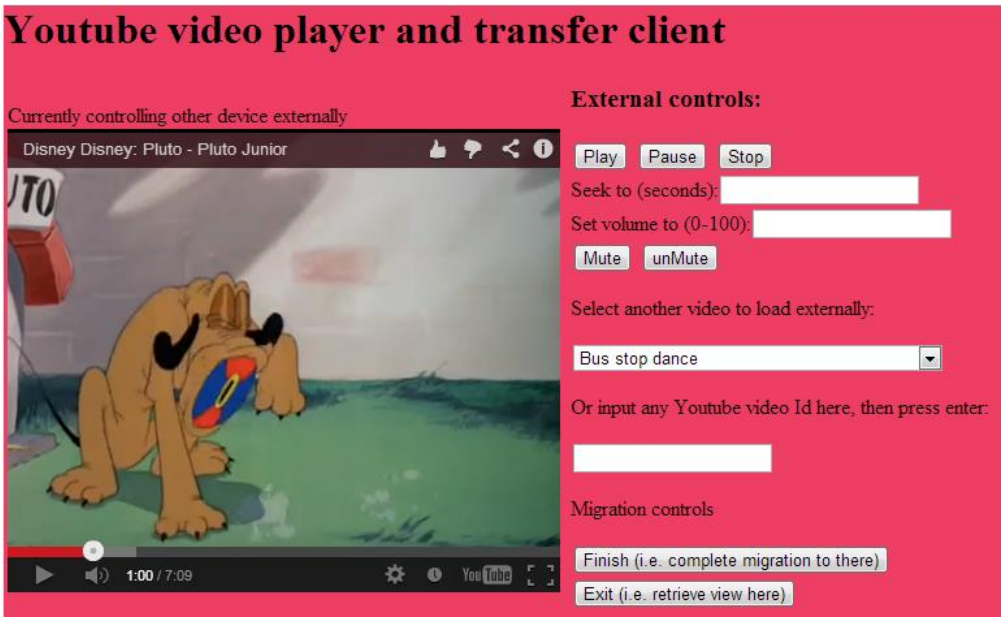

Figure 43. YouTube external view demo. Controlling view with external controls

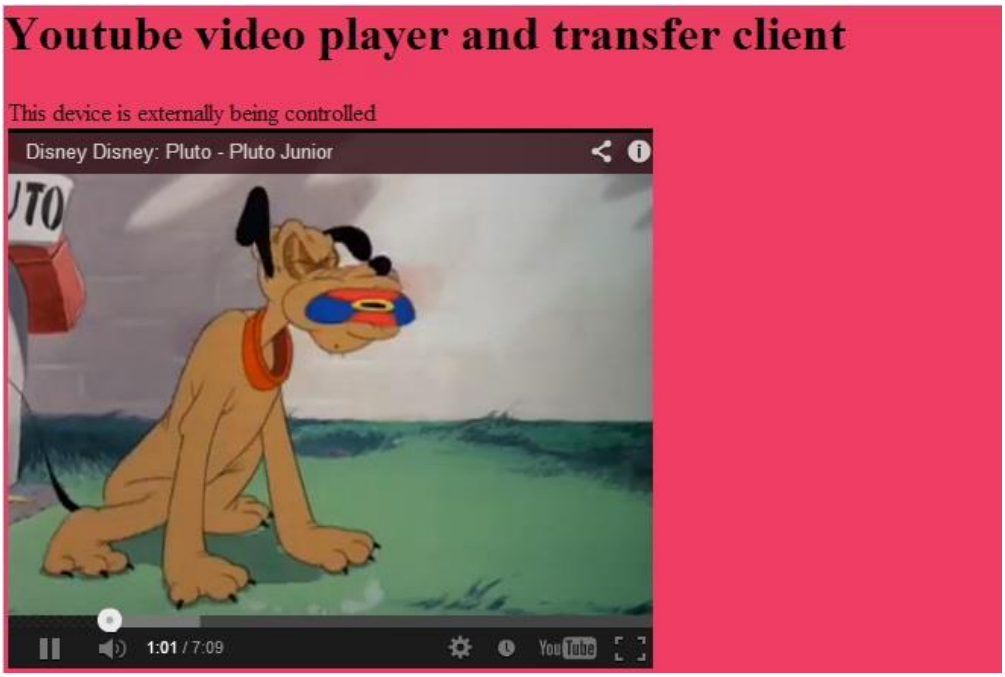

Figure 44. YouTube external view demo. Controlled view. No controls.

Figure 42 shows a user choosing to do an immediate external view migration of an ongoing YouTube video playback from his "source" device to his "target"-device. Using WebMP, the "source" device creates a peer connection with the "target" and transfers the state. The "target" then launches *externally,* i.e. continues the video playback, but in a "controlled" view. Figure 43 shows the "controlling" view at the "source", while Figure 44 shows the "controlled" view at the "target". The user is now able to control the video playback at the "target" from his mobile device, as well as finish or exit the migration session at any time.

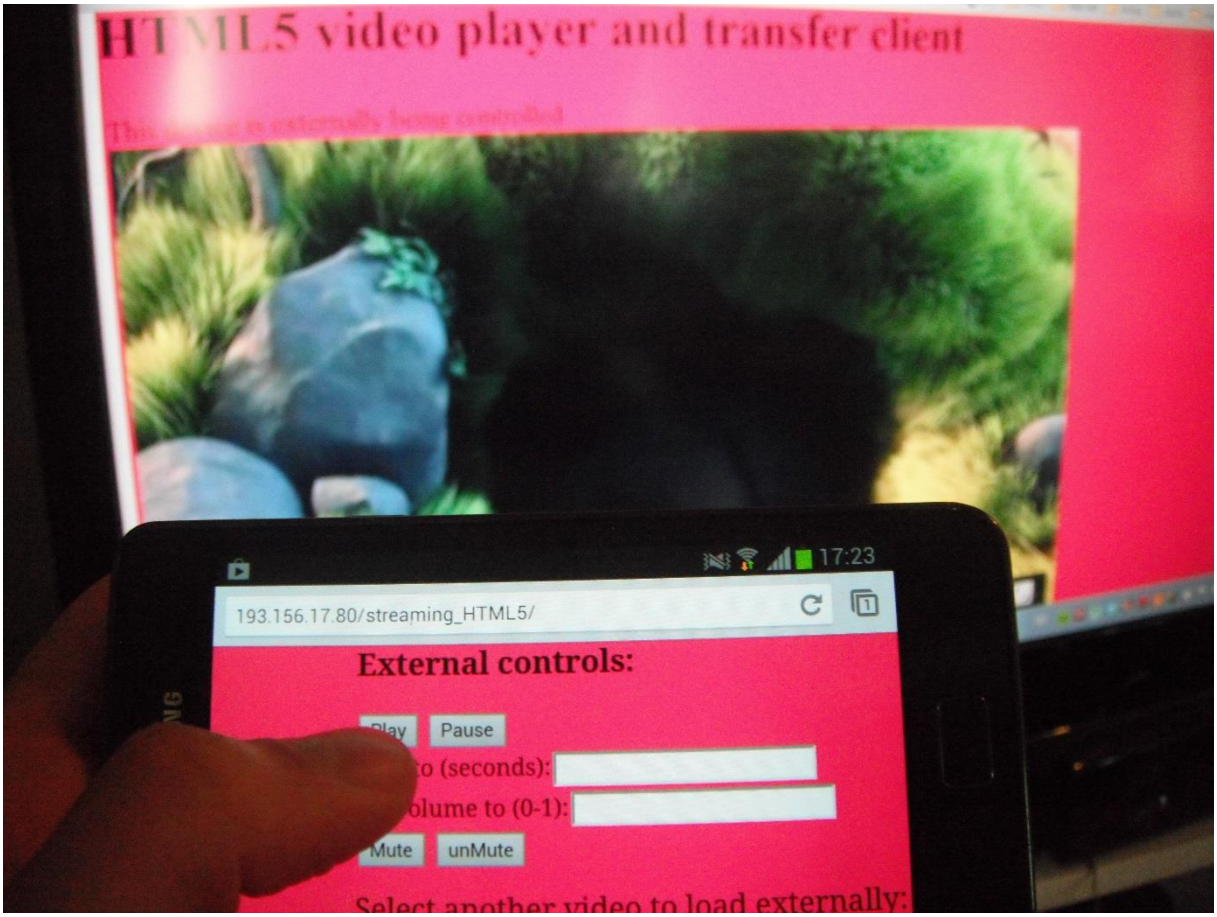

Figure 45. YouTube external view scenario. Controlling playback on TV via smartphone.

Figure 45 shows a typical scenario in which external view would be a useful feature – the user controlling an HTML5 video playback at the TV from his smartphone, just like a remote control.

8.5 Results

With our proof of concept and demo applications, we have showcased a generic, cross-device, cross-domain session mobility platform for both web applications and browser extensions, with similar functionality and behavior as the proposed solution design.

Developing the different demo applications worked more or less in a “plug-and-play”-fashion, by including the exact same generic client-side “migration scripts” (WebMC) on every application, and implementing the required application-specific functions. We absolutely believe this can be done on existing web applications, as it requires little (in the case of RTC-apps) or none modification of the existing application code, working more or less as an *extension* to the original application. Also, by having the app developers define the required functions, we provide them with complete freedom when it comes to how they want to apply the migration functionalities, both when it comes to behavior and appearance.

As of today, the WebMP only works for Chrome. But this is only temporarily, should we believe the WebRTC project. Once WebRTC is finalized, we can expect interoperability between all the major browsers, and possibly other platforms as well (WebRTC also has native APIs). Thus, WebRTC stands out as a potential interesting solution base not only for a proof of concept.

During testing, the proof of concept performed great, producing very little delay in our demo applications. This is especially important in RTC-apps, such as the videochat application. However, we haven’t conducted any performance evaluation of our implementation, so we can’t make any well-informed conclusions about its performance. Hopefully, as promised by Node, the implementation should perform and scale well to more intense workloads as well.

9 Conclusion

Today, cross-device capabilities has become the de facto standard among most applications, enabling users to access an application and resume his/her previous session from almost any device. This has become a reality due to the ubiquitous nature of Internet connectivity, and is realized by utilizing technologies such as Cloud Computing, and cross-platform development and user interface tools such as HTML5. Additionally, low end devices continuously become more powerful, enabling them to do much more than before. While this allows you to run the same application on different devices, there still exists no widespread solution providing the ability to transfer an ongoing application from one device to another, continuing the ongoing session immediately. Since the users we've asked have considered such functionality to be of great value, this is definitely something worth exploring. Hence, one of the main goals of this project has been to investigate and determine the feasibility of a cross-device session mobility platform.

A truly generic cross-device session mobility platform should work with any application, irrespective of the type of the application and the type of terminal it is running on. This becomes a challenge, considering that the nature of an application may vary a lot from application to application. We have identified five application categories in our effort to categorize them based on their functionality: browsing, business, multimedia, communication and gaming applications. Out of these five, users we've asked considered all categories except from video gaming as possible beneficiaries of session mobility – some more than others.

Additionally, we've identified three application type implications that need to be considered; whether an application is real-time or not, the set of protocols it uses, and its architecture. When it comes to real-time communication applications, it's important to remember that the remote endpoint usually is a person. Thus, we need to take into account his/her user experience as well when designing the session mobility mechanism.

Before designing our solution, we needed to analyze and evaluate the various approaches used in related work and existing solutions, as these would serve as our inspiration. Our goal was to find and propose a feasible solution to the challenge of creating a generic, cross-device session mobility platform that would also satisfy the requirements we derived from our user study.

When we started the theoretical approach of our solution, we focused on the actual migration procedure, where we stood between a live migration and cold migration approach. In a live migration, the goal is to retain the session throughout a migration. Here, a proxy entity was introduced, which acted on the behalf of its connected devices/applications. This way, the migration was kept transparent to the other

endpoint, but as a result could lead to the need for performing real-time content adaptation, or live transcoding, which we've deemed undesirable. During our comparison of the two alternatives, we concluded that the need for retaining a session is only justifiable when considering real-time communication applications, because of the remote user's experience. For other applications, a workaround could simply be to stop the application on the source device, transfer the state of the application session to the target device, and then launch the application there, passing the transferred state as an argument. This is the approach of cold migration. We still needed to solve the problem with real-time communication applications, though.

Our proposed solution, the Migration Platform, is realized as a centralized peer-to-peer architecture. It consists of two entities; the Migration Server (MS) and the Migration Client (MC). Together, the MS and MC will provide any application with cross-device session mobility capabilities, including both the "external view" (i.e. a partial migration where the application view on one device is controlled from another device) and "full migration" features.

The MS becomes the connection point for all the MCs, its main responsibilities being to provide connectivity, device discovery, and keep track of associations, as well as aid in setting up peer connections between devices upon a migration.

The MC is software that has to be installed on each device. It is responsible for connecting to the MS via a signaling module, as well as executing the actual migration via a Peer Connection (PC) module, by establishing a direct data channel between source and target device. The state of the application about to be migrated is transferred over the data channel, with little or none restrictions to the size or structure of the state object. In order to launch applications not currently running on the device, the MC also has to be able to interface to the device's hardware, meaning it has to be native.

The applications that are to utilize the capabilities provided by the MP need to implement an interface to the MC, as well as providing a user interface for the migratory functions, enabling the user to trigger migration functions from the application. This way, the application can invoke migration functions provided by the MC, and the MC can invoke the application-specific functions specified in the interface. The most important application-specific functions we require the application to implement for the migration procedure, are a state retrieval function, a stateful launch function, and functions for sending and executing actions when in an external view session. During a normal migration scenario, the state retrieval function should return a state object, representing the current state of the application on the source device. The state is transferred to the target device, where it will be passed as an argument upon launching the application. This stateful launch should ensure that the application session resumes from where it left off in the source device.

The behavior of the MP (i.e. the peer connection setup, session transfer etc.) is kept transparent to the application developer, who only has to implement the interface, making our solution work in a 'plug-and-play'-fashion. By having the application developers implement the required functions, we provide them with complete freedom

when it comes to how they want to apply the migration functionalities, both when it comes to behavior and appearance.

The main limitation of our proposed scheme is the inability to seamlessly migrate real-time communication applications. But it is a limitation that can be overcome with slight application modification. The suggested workaround is to have the migrating peer send an application-specific message to the remote peer upon a migration request, notifying it about the upcoming migration. As such, the remote peer can react appropriately, by e.g. enter a wait state and communicate this to the end user, maintaining his/her user experience. We argue such a modification should be minor, and thus acceptable. This way, our solution is able to work with both real-time and non-real time applications, rendering it (almost) generic.

Compared to DIAL, which was mentioned as perhaps the most complete available solution for session mobility, we offer a number of improvements, on top of the already explained session mobility. Instead of requiring all devices to be connected to the same LAN, our solution is cross-domain, meaning devices are able to communicate as long as they have Internet connectivity. Our solution also offers bidirectional migration (meaning that the transfer can occur from any-to-any device, unlike the 1st screen-to-2nd screen scheme DIAL uses). Additionally, our solution offers the "external view" feature out of the box, while such functionality is out of the scope of DIAL.

With our proof of concept, the goal was to showcase the viability of our proposed solution design, with the help of a couple of demo applications. We implemented it in the web environment, providing a generic cross-device (Android/PC) session mobility platform for web applications and extensions. We chose the web environment due to the promising future of HTML5, as well as the fact that HTML5 makes it easy to deploy cross-device applications. Our implementation was heavily based on the bleeding edge technology WebRTC, as well as the cutting edge technologies Node and Socket.IO, all purely JavaScript-based. WebRTC is an API currently being drafted that enables web browsers with real-time communication capabilities.

Conforming to the solution design, the proof of concept platform consisted of two entities, the Web Migration Server (WebMS) and the Web Migration Client (WebMC). The WebMS was implemented as a Socket.IO server entity, running on Node. The WebMC connected to the WebMS via a Socket.IO-based signaling module. The Peer Connection module of the WebMC was built around WebRTC. We utilized its PeerConnection and DataChannel APIs to create and establish the peer connection, as well as the data channel for session transfer. The logic of the solution was implemented in JavaScript.

The applications that were to utilize the capabilities provided by our proof of concept needed to include the generic scripts (thus implementing the functions of the "WebMC"), and implement some application-specific functions. We implemented four demo applications; two video streaming applications (HTML5 and YouTube) that implemented both "full migration" and "external view", a videochat application (also built in WebRTC)

and a Chrome browsing extension. The videochat application had to implement an additional notification message, which we showed could be done as a minor client-side modification, thus strengthening our argument of the viability for real-time communication applications working with our solution. Having the proof of concept work with all these different types of application, even third-party APIs (YouTube and WebRTC) clearly showcases the generic property of our solution.

The Migration Client in our proof of concept differed in two ways from the proposed solution. Firstly, we didn't implement any native module that would allow a background service running in OS level in order to launch an application on the target device upon migration. Instead, it required the target application to be up and running on the target device. Secondly, we included a separate Migration Client for each web application, not a common one. This was due to security implications with Chrome. Though, for a proof of concept we argue these sidesteps are acceptable, as we still succeeded in proving the viability of a generic, cross-device session-mobility platform.

With our proposed solution design, as well as our implemented proof of concept, showing its viability, we argue that we have proved that a generic, cross-device session mobility platform is both possible and feasible, but not without a certain level of application modification, which we believe is a reasonable requirement. Working with a bleeding edge technology like WebRTC has been both very challenging and interesting, and by looking at the developer's feedback, critics' reviews and the general enthusiasm about it in the technology world, we do believe that after its full development and hopefully widespread adoption by web-application developers, our solution's value and applicability will increase even more.

10 Future Work

There are several things we have identified as opportunities for future work.

First, we did not perform any thorough security assessment of our proposed solution design, and as such identify this as perhaps the most critical thing that needs to be done. First of all, this entails doing a study to identify the security vulnerabilities and risks in regards to our solution. When this is identified, the necessary security measures, such as implementing appropriate user/device authentication and encryption must be implemented.

We have also identified a couple of additional, potentially useful, features which we did not include in our use cases, requirements and solution design. These are:

- The ability to identify the connected devices that have not installed the application currently running on the source device, but are capable of running it. Thus, if the requested application is not installed on the target device, the solution needs to point out a way for the device to download and install it before the migration can take place. This is inspired by a similar feature in DIAL (53). Though, before including such a functionality, we need to evaluate its viability and usability, as it will require considerably more overhead and context management.
- Trigger management. With trigger management, the solution should be able to decide when a migration should occur, and make it happen automatically, thus minimizing user effort and hopefully enhancing the user experience. Consider e.g. an automatic migration of a movie playback from the TV in the living room to the TV in the kitchen once the user moves. This is inspired by a similar module in the OPEN project (51).

Before implementing such additional features, however, we believe it's more attractive to start implement the solution as-is, e.g. by starting with the most popular platforms. Or, we could focus on enhancing the proof of concept.

When it comes to our proof of concept, the obvious further work will be to make it interoperable between all the major browsers once WebRTC is finalized and/or allows for this. At this point, one should make an assessment of the applicability/usability of WebRTC's use outside the web environment as well, as WebRTC has a native API.

Although the demo applications performed well, we have yet to execute any performance evaluation of the proof of concept. Unfortunately, we didn't have the necessary equipment to do any thorough testing on the mobile platform either, except assuring it did work cross-device. Thus, a performance evaluation (and/or simply a bug test) focusing on the mobile platform could be desirable.

11 References

1. **Shacham, R, et al.** Session Initiation Protocol (SIP) Session Mobility. *ietf.org.* [Online] Network Working Group. [Cited: 3 13, 2013.] http://tools.ietf.org/html/rfc5631.

2. **Johansson, D.** *Session mobility in multimedia services enabled by the cloud and peer-to-peer paradigms.* 2011.

3. **Højgaard-Hansen, Kim, Nguyen, Huan Cong and Schwefel, Hans-Peter.** *Session mobility solution for client-based application migration scenarios.* s.l. : 2011 Eighth International Conference on Wireless On-Demand Netwrok Systems and Services, 2011.

4. **Google.** Google play. [Online] Google. [Cited: 1 25, 2013.] https://play.google.com/store.

5. **Apple.** Apple. *From the app store.* [Online] Apple. [Cited: 1 25, 2013.] http://www.apple.com/iphone/from-the-app-store/.

6. **Hanssen, Øystein Wethe.** *Actors and ecosystem in the cloud computing market.* Trondheim : NTNU, 2012.

7. **Tantow, Martin.** Cloud Computing: Current Market Trends and Future Opportunities. *CloudTimes.* [Online] CloudTimes. [Cited: 05 29, 2013.] http://cloudtimes.org/2011/06/22/cloud-computing-its-current-market-trends-and-future-opportunities/.

8. **DropBox.** DropBox. *DropBox.* [Online] DropBox. [Cited: 5 29, 2013.] https://www.dropbox.com/.

9. **Google.** Google AppEngine. *Google AppEngine.* [Online] Google. [Cited: 5 29, 2013.] https://developers.google.com/appengine/.

10. **Wikipedia.** HTML5. [Online] Wikipedia. [Cited: 2 12, 2013.] http://en.wikipedia.org/wiki/HTML5.

11. **Lardinois, Frederic.** Survey: Most Developers Now Prefer HTML5 For Cross-Platform Development. *Tech Crunch.* [Online] Tech Crunch. [Cited: 03 13, 2013.] http://techcrunch.com/2013/02/26/survey-most-developers-now-prefer-html5-for-cross-platform-development/.

12. **Wikipedia.** DOM_events. *Wikipedia.* [Online] [Cited: 4 3, 2013.] http://en.wikipedia.org/wiki/DOM_events.

13. **WebSocket.org.** About HTML5 WebSockets. *WebSocket.org.* [Online] Kaazing. [Cited: 4 24, 2013.] http://www.websocket.org/aboutwebsocket.html.

14. **Sutherland, Ed.** Gartner: more than half of mobile apps will be HTML5/native hybrids by 2016. *idownloadblog.* [Online] 2 4, 2013. [Cited: 3 20, 2013.] http://www.idownloadblog.com/2013/02/04/gartner-mobile-apps-2016/.

15. **Ness.** Gartner Hype Cycle Report Predicts HTML5 Still Years Away. *Ness Blog.* [Online] 8 29, 2012. [Cited: 2 13, 2013.] http://blog.ness.com/spl/bid/81824/Gartner-Hype-Cycle-Report-Predicts-HTML5-Still-Years-Away.

16. **W3C.** TAKE CONTROL — YOUR WEB, YOUR LOGO. *W3C.* [Online] W3C. [Cited: 5 15, 2013.] http://www.w3.org/html/logo/.

17. **Mozilla.** Firefox OS. *Mozilla Developer Network.* [Online] Mozilla. [Cited: 2 22, 2013.] https://developer.mozilla.org/en/docs/Mozilla/Firefox_OS.

18. **Tizen.** Tizen. *Tizen.* [Online] Linux Foundation. [Cited: 2 22, 2013.] https://www.tizen.org/.

19. **Tode, Chantal.** HTML5-based mobile operating systems may change the face of mobile marketing. *MobileMarketer.* [Online] 7 6, 2012. [Cited: 2 15, 2013.] http://www.mobilemarketer.com/cms/news/software-technology/13254.html.

20. **Tuominen, Timo.** Metro on Windows 8 with "Native HTML5". *futurice blog.* [Online] 3 7, 2012. [Cited: 2 16, 2013.] http://blog.futurice.com/metro-on-windows-8-with-native-html5.

21. **Samsung.** HTML 5 Specification. *samsungdforum.* [Online] [Cited: 3 12, 2013.] http://www.samsungdforum.com/upload_files/files/guide/data/html/html_3/reference/browser%20spec_html5.html.

22. **Daoust, Francois.** *Adopting HTML5 for Television: Next Steps.* s.l. : W3C.

23. **w3schools.com.** Browser Statistics. *w3schools.com.* [Online] [Cited: 5 23, 2013.] http://www.w3schools.com/browsers/browsers_stats.asp.

24. **Google.** Google Sync. *www.google.com.* [Online] Google. [Cited: 3 13, 2013.] http://www.google.com/sync/index.html.

25. **Mozilla.** Firefox Sync - Take your bookmarks, tabs and personal information with you. *Mozilla support.* [Online] Mozilla. [Cited: 3 13, 2013.] http://support.mozilla.org/en-US/kb/firefox-sync-take-your-bookmarks-and-tabs-with-you.

26. **Opera.** Opera Link. *www.opera.com.* [Online] Opera. [Cited: 3 13, 2013.] http://www.opera.com/link.

27. **Google.** Sync. *The Chromium Projects.* [Online] Google. [Cited: 3 13, 2013.] http://www.chromium.org/developers/design-documents/sync.

28. **Galbraith, Ben.** Coping with Over Four Hundred Devices: How Netflix Uses HTML5 to Deliver Amazing User Interfaces. *FunctionSource.* [Online] FunctionSource. [Cited: 2 19, 2013.] http://functionsource.com/post/netflix-feature.

29. **Jacobsen, Daniel.** The Netflix Tech Blog. *Embracing the Differences : Inside the Netflix API Redesign.* [Online] Netflix. [Cited: 2 1, 2013.] http://techblog.netflix.com/2012/07/embracing-differences-inside-netflix.html.

30. **Galbraith, Ben.** Function Source. *Coping with Over Four Hundred Devices: How Netflix Uses HTML5 to Deliver Amazing User Interfaces.* [Online] Fucntion Source, 2011. [Cited: 1 31, 2013.] http://functionsource.com/post/netflix-feature.

31. **Strickland, Jonathan.** How Google Docs Works. *How stuff works.* [Online] [Cited: 3 1, 2013.] http://computer.howstuffworks.com/internet/basics/google-docs5.htm.

32. **Russell, Jon.** Microsoft begins to integrate its enterprise-focused Lync service with Skype. *thenextweb.* [Online] TNW. [Cited: 3 6, 2013.] http://thenextweb.com/microsoft/2013/04/17/microsoft-begins-to-integrate-its-enterprise-focused-lync-service-with-skype/.

33. **Microsoft.** Lync Top Features. *office.microsoft.com.* [Online] Microsoft. [Cited: 3 11, 2013.] http://office.microsoft.com/en-001/lync/microsoft-lync-top-features-video-conferencing-and-instant-messaging-FX103789488.aspx.

34. —. Microsoft Lync How-to Guide. [Online] Microsoft. [Cited: 3 12, 2013.] http://ecenter.custhelp.com/ci/fattach/get/270619/1344868514/redirect/1/session/L2F2LzEvdGltZS8xMzY5ODI5ODEwL3NpZC96NUJKT29ybA==/filename/Microsoft%20Lync%20How%20To%20Guide.pdf.

35. **Wikipedia.** SIP Trunking. *Wikipedia.* [Online] Wikipedia. [Cited: 3 13, 2013.] http://en.wikipedia.org/wiki/SIP_Trunking.

36. **Rosenberg, J, et al.** SIP: Session Initiation Protocol. *ietf.org.* [Online] Network Working Group. [Cited: 3 13, 2013.] http://www.ietf.org/rfc/rfc3261.txt.

37. **Microsoft.** Lync Server 2013. *Lync.* [Online] Microsoft. [Cited: 3 14, 2013.] http://office.microsoft.com/en-us/lync/lync-server-2013-features-video-conferencing-and-instant-messaging-FX103789592.aspx.

38. **Warren, Tom.** Microsoft explains Xbox One cloud gaming in an effort to justify online requirement. *The Verge.* [Online] The Verge. [Cited: 4 2, 2013.] http://www.theverge.com/2013/5/24/4361730/xbox-one-cloud-gaming-part-of-online-requirement.

39. **Wikipedia.** Cloud gaming. *Wikipedia.* [Online] Wikipedia. [Cited: 4 4, 2013.] http://en.wikipedia.org/wiki/Cloud_gaming.

40. **Nvidia.** NVIDIA GRID. *Nvidia.* [Online] Nvidia. [Cited: 5 29, 2013.] http://www.nvidia.com/object/cloud-gaming.html.

41. **CiiNOW.** CiiNOW. *CiiNOW.* [Online] CiiNOW. [Cited: 5 19, 2013.] http://www.ciinow.com/.

42. **CiiNow.** Cloud gaming technology. *CiiNow.* [Online] CiiNOW. [Cited: 4 18, 2013.] http://www.ciinow.com/cloud-gaming-service-technology/.

43. **VirtualMeshTest.** Cold Migration and Live Migration. *wiki.umiacs.umd.edu.* [Online] VirtualMeshTest. [Cited: 4 2, 2013.] https://wiki.umiacs.umd.edu/VirtualMeshTest/index.php/Cold_Migration_and_Live_Migration.

44. **Mate, Sujeet, Chandra, Umesh and Curcio, Igor D.** *Movable-multimedia: session mobility in ubiquitous computing ecosystem.* 2006.

45. **DoCoMo Euro-Labs.** *Session Initiation Protocol (SIP) Session Mobility.* s.l. : Network Working Group , 2009.

46. **Sparks, R, Johnston, A and Petrie, D.** Session Initiation Protocol (SIP) Call Control - Transfer. *tools.ietf.org.* [Online] [Cited: 3 18, 2012.] http://tools.ietf.org/html/rfc5589.

47. **Adeyeye, Michael, Ventura, Neco and Foschini, Luca.** *Converged multimedia services in emerging Web 2.0 session mobility scenarios.* 2011.

48. **Barisch, Marc, Kögel, Jochen and Meier, Sebastian.** *A Flexible Framework for Complete Session Mobility and Its Implementation.* Stuttgart : Institute of Communication Networks and Computer Engineering, Universität Stuttgart, 2009.

49. **Canfora, G., et al.** *Proxy-based Hand-off of Web Sessions for User Mobility.* 2005.

50. **Partners, Open.** *OPEN Project .* s.l. : NEC Europe , 2009.

51. **Nickelsen, Anders, et al.** *OPEN: Open pervasive environments for migratory interactive services.* s.l. : iiWAS2010 Proceedings, 2010.

52. **Wikipedia.** SOCKS. *Wikipedia.* [Online] Wikipedia. [Cited: 2 26, 2013.] http://en.wikipedia.org/wiki/SOCKS#SOCKS5.

53. **Netflix.** *DIAL - Discovery and Launch protocol specification. Version 1.6.4.* s.l. : Netflix, 2012.

54. **Roettgers, Janko.** The story behind DIAL: How Netflix and YouTube want to take on AirPlay. *GigaOM.* [Online] GigaOM. [Cited: 2 15, 2013.] http://gigaom.com/2013/01/23/dial-open-airplay-competitor/.

55. **Åhlund, Andreas, et al.** *Context-aware Application Mobility Support in Pervasive Computing Environments.* s.l. : Proceedings of the 6th International Conference on Mobile Technology, Application & Systems, 2009.

56. **Wikipedia.** Reflection (computer programming). *Wikipedia.* [Online] Wikipedia. [Cited: 3 7, 2013.] http://en.wikipedia.org/wiki/Reflection_(computer_programming).

57. **G. Camarillo, Ed.** RFC5694: Peer-to-Peer (P2P) Architecture: Definition, Taxonomies, Examples, and Applicability. [Online] 2009. [Cited: 4 7, 2013.] http://tools.ietf.org/html/rfc5694.

58. **Wikipedia.** Peer-to-Peer. *Wikipedia.* [Online] [Cited: 4 7, 2013.] http://en.wikipedia.org/wiki/Peer-to-peer.

59. **Netflix.** About the Registry. *DIAL.* [Online] Netflix, Google. [Cited: 5 22, 2013.] http://www.dial-multiscreen.org/dial-registry.

60. **Mudge, JT.** Native App vs. Mobile Web App: A Quick Comparison. *Six Revisions.* [Online] Six Revisions. [Cited: 5 22, 2013.] http://sixrevisions.com/mobile/native-app-vs-mobile-web-app-comparison/.

61. **Wikipedia.** Heartbeat message. *Wikipedia.* [Online] Wikipedia. [Cited: 5 23, 2013.] http://en.wikipedia.org/wiki/Heartbeat_message.

62. **Google.** WebRTC. *WebRTC.* [Online] W3C WebRTC Working Group. [Cited: 3 1, 2013.] http://www.webrtc.org/.

63. **Rauch, Guillermo.** Socket IO. *Scoket IO.* [Online] LearnBoost. [Cited: 4 10, 2013.] http://socket.io/.

64. **Joyent.** Node JS. *Node JS.* [Online] Joyent. [Cited: 5 1, 2013.] http://nodejs.org/.

65. **Bergkvist, Adam, et al.** WebRTC 1.0: Real-time Communication Between Browsers. *dev.w3.org.* [Online] W3C WebRTC Working Group. [Cited: 3 25, 2013.] http://dev.w3.org/2011/webrtc/editor/archives/20130322/webrtc.html.

66. **Dutton, Sam.** Getting Started with WebRTC. *html5rocks.* [Online] 7 23, 2012. [Cited: 3 25, 2013.] http://www.html5rocks.com/en/tutorials/webrtc/basics/.

67. **Pfeiffer, Silvia.** Implementing Video Conferencing in HTML5. [Online] 2012. [Cited: 4 3, 2013.] http://html5videoguide.net/presentations/WebDirCode2012/#page3.

68. **Uberti, J and Jennings, C.** Javascript Session Establishment Protocol. *tools.ietf.org.* [Online] Network Working Group. [Cited: 4 9, 2013.] http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03.

69. **Handley, M, Jacobsen, V and Perkins, C.** SDP: Session Description Protocol. [Online] Network Working Group. [Cited: 4 9, 2013.] http://tools.ietf.org/html/rfc4566.

70. **Nandakumar, S and Jennings, C.** SDP for the WebRTC. [Online] Network Working Group. [Cited: 4 11, 2013.] http://tools.ietf.org/id/draft-nandakumar-rtcweb-sdp-01.html.

71. **Uberti, J and Jennings, C.** Javascript Session Establishment Protocol. [Online] Network Working Group, 2 25, 2013. [Cited: 4 15, 2013.] http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03.

72. **Rosenberg, J, et al.** Session Traversal Utilities for NAT (STUN). [Online] Network Working Group . [Cited: 5 30, 2013.] http://tools.ietf.org/html/rfc5389.

73. **Mahy, R, Matthews, P and Rosenberg, J.** Traversal Using Relays around NAT (TURN):Relay Extensions to Session Traversal Utilities for NAT (STUN). [Online] Network Working Group. [Cited: 5 30, 2013.] http://tools.ietf.org/html/rfc5766.

74. **Rosenberg, J.** Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. [Online] Network Working Group. [Cited: 5 30, 2013.] http://tools.ietf.org/html/rfc5245.

75. **VOIP-info.** STUN. *Voip-Info.org.* [Online] VIOP-info. [Cited: 4 2, 2013.] http://www.voip-info.org/wiki/view/STUN.

76. **Google.** Free open source implementation of TURN and STUN Server. *code.google.com.* [Online] Google. [Cited: 4 2, 2013.] https://code.google.com/p/rfc5766-turn-server/.

77. —. Important Concepts. *developers.google.com.* [Online] Google. [Cited: 4 2, 2013.] https://developers.google.com/talk/libjingle/important_concepts.

78. **Crockford, D.** The application/json Media Type for JavaScript Object Notation (JSON). [Online] Network Working Group. [Cited: 4 10, 2013.] http://tools.ietf.org/html/rfc4627.

79. **W3C WebRTC Working Group.** Interop Notes. *WebRTC.* [Online] W3C WebRTC Working Group. [Cited: 5 2, 2013.] http://www.webrtc.org/interop.

80. **Google.** V8 JavaScript Engine. *v8.* [Online] Google. [Cited: 5 3, 2013.] https://code.google.com/p/v8/.

81. **Paul, Geo.** Beginner's Guide To Node.Js (Server-Side JavaScript). *Hongkiat.com.* [Online] Hongkiat. [Cited: 4 5, 2013.] http://www.hongkiat.com/blog/node-js-server-side-javascript/.

82. **Walsh, David.** WebSocket and Socket.IO. *The David Walsh Blog.* [Online] David Walsh. [Cited: 5 3, 2013.] http://davidwalsh.name/websocket.

83. **LearnBoost Labs.** FAQ. *Socket IO.* [Online] LearnBoost Labs. [Cited: 5 4, 2013.] http://socket.io/#faq.

84. **LearnBoost.** Configuring Socket.IO. *GitHub.* [Online] LearnBoost. [Cited: 5 5, 2013.] https://github.com/LearnBoost/Socket.IO/wiki/Configuring-Socket.IO.

85. **Wikipedia.** Socket.IO. *Wikipedia.* [Online] Wikipedia. [Cited: 5 3, 2013.] http://en.wikipedia.org/wiki/Socket.io.

86. **w3schools.** HTML5 Video. *w3schools.* [Online] w3schools. [Cited: 5 10, 2013.] http://www.w3schools.com/html/html5_video.asp.

87. **Google.** swfobject. *code.google.com.* [Online] Google. [Cited: 5 16, 2013.] https://code.google.com/p/swfobject/.

88. —. YouTube JavaScript Player API Reference. *developers.google.com.* [Online] Google. [Cited: 5 11, 2013.] https://developers.google.com/youtube/js_api_reference.

89. —. What are extensions? *developer.chrome.com.* [Online] Google. [Cited: 5 3, 2013.] http://developer.chrome.com/extensions/index.html.

90. **LearnBoost.** Configuring Socket.IO. *GitHub.* [Online] LearnBoost. [Cited: 5 22, 2013.] https://github.com/LearnBoost/Socket.IO/wiki/Configuring-Socket.IO.

91. **Kaltura.** Best Practices For Multi-Device Transcoding. *Kaltura.* [Online] Kaltura. [Cited: 4 3, 2013.] http://knowledge.kaltura.com/best-practices-multi-device-transcoding.

92. **Wikipedia.** Streaming media. *Wikipedia.* [Online] Wikipedia. [Cited: 4 7, 2013.] http://en.wikipedia.org/wiki/Streaming_media.

93. **FFmpeg.** FFmpeg. *FFmpeg.* [Online] FFmpeg. [Cited: 4 7, 2013.] http://www.ffmpeg.org/.

94. **PeerJS.** PeerJS. *PeerJS.* [Online] PeerJS. [Cited: 5 2, 2013.] http://peerjs.com/status.

95. **Google.** FAQ. *developers.google.com.* [Online] Google. [Cited: 5 29, 2013.] https://developers.google.com/chrome/mobile/docs/faq.

96. **Wikipedia.** Application Programming Interface. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Application_programming_interfaces.

97. —. Web Applications. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Web_application.

98. **Anne van Kesteren, Simon Pieters.** HTML5 differences from HTML4. *W3C.* [Online] http://www.w3.org/TR/html5-diff/.

99. **Wikipedia.** HTML5 in mobile devices. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/HTML5_in_mobile_devices.

100. **Rowinski, Dan.** HTML5: Don't Believe the Hype Cycle. *ReadWrite.* [Online] August 21, 2012. http://readwrite.com/2012/08/21/html5-ready-for-prime-time-dont-believe-the-hype-cycle.

101. —. Mobile Devs Interested in Google Over Facebook for Social Mobile Apps. *ReadWrite.* [Online] March 19, 2012. http://readwrite.com/2012/03/19/mobile-devs-increasingly-inter.

102. **Google.** Sync. *The Chromium Projects.* [Online] http://www.chromium.org/developers/design-documents/sync.

103. **Uno, Rave.** Peer-to-Peer Vs. Client Server Networks. *Buzzle.* [Online] 2011. http://www.buzzle.com/articles/peer-to-peer-vs-client-server-networks.html.

104. **Wikipedia.** SOCKS. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/SOCKS.

12 Appendix

12.1 User survey

We used SurveyMonkey (http://www.surveymonkey.com/), a free online survey software and questionnaire tool, to create the user survey.

The survey can be found here (in Norwegian):
http://www.surveymonkey.com/s/92TRBZQ

12.2 Content adaptation analysis

In this section, we will investigate the feasibility of performing real-time content adaptation in the proxy-based live migration approach, discussed in section 6.2.2. Here, a proxy is placed between the application clients and the other endpoint (i.e. the application server or the remote peer). This proxy is responsible for orchestrating a migration of an ongoing application session, while at the same time keeping the other endpoint transparent of this migration. Hence, the proxy may have to perform real-time content adaptation of the data being sent both from and to the new client device, as this device may not have the same capabilities as the previous device did.

In this section we will first introduce the reader to what this content adaptation actually entails. What type of content adaptation is needed, and how is it done? Finally, we will analyze the feasibility of including such a real-time content adaptation in a solution design.

12.2.1 Introduction to content adaptation, or transcoding

Formats and codecs

In order for a file to be interpreted, it needs to follow rules that describe how data should be stored in the file. The rules are determined by a *container format*, while the file itself is both encoded and decoded (interpreted) by a *codec*.

The *container format*, tells you which type of container the file uses as a transport medium. The format/standard determines how a valid bitstream has to look like. This may be defined by standards such as AVI (Audio Video Interleaved), .mpg, .mov, etc. Thus, the container format determines the way the information is stored and delivered.

A container family is used to provide a single file format to the user, even though the container actually consists of multiple, different files. A popular family of containers is found for use with multimedia file formats. Here, audio and video streams may be combined (multiplexed) in the same container. Later, these streams can be individually selected (demultiplexed) for decoding.

The *codec* (compressor, decompressor), such as x264, XviD and DivX, is the algorithms used to compress the information for delivering and storing, and to decompress the information for displaying it. I.e. the codec is software that does the actual compression/decompression of the information in accordance with a container format standard. For a certain format, there may exist many codecs capable of encoding/decoding it. While their algorithms for encoding/decoding differ, they all should produce a valid format bitstream that can be decoded by any of the codecs that supports decoding of that format.

Focus area

Both codecs and formats play an important role for determining the file size and quality of the file. Thus, different codecs and container formats may be applicable for different devices, as e.g. screen sizes and bandwidth may vary a lot. This results in different support of formats and codecs among the devices, and thus the need for conversion (transcoding) between these.

We have identified the need for real-time content adaptation (or live transcoding) in a live migration solution. What this entails is the need for real-time format transcoding of incoming (and outcoming, in the case of RTC apps.) data, in order to meet the requirements of the receiving entities (e.g. the endpoints of a video conference).

We argue data traffic from streaming and real-time communication applications mostly will be the subject to content adaptation. Hence, multimedia data will be the main subject to transcoding. But how is transcoding of multimedia in a cross-device environment actually done?

Best practices for multi-device transcoding (related to streaming video)

Today, many streaming services are available across devices. This creates a challenge for the service provider: to find the right balance between bit rate and resolution as it relates to the end user's connection speed and system ability. Here, there are many variables to account for. Of the most obvious, are:

- Device and Screen size
- Internet Service Provider Connection Speed or Cap
- Bandwidth available through the wireless router due to distance or usage
- Network traffic to and from the server hosting the video
- The CPU and GPU abilities on the playing device
- Browser brand and version
- Available plugins such as Flash and Silverlight
- Other programs running in the background

(91).

The overall goal will be to address the above variables in a way that provides immediate, uninterrupted, and smooth playback in the highest quality possible. Today, it has become more and more common to address this challenge with so-called *adaptive streaming*.

Adaptive streaming

Transcoding always starts with a master file. The transcode can never be of a better quality than the master, and it's often too big to even play back on most computers and transcode from (consider an uncompressed 4k piece of video). Thus, the master file must be converted to different bit rates that may even scale down in resolution (see Figure 46) - adaptive bit rates.

Adaptive streaming is a technique for detecting and adjusting to a user's bandwidth capabilities in real-time. It works by having multiple bit rates available that the player or server can pull based on the user's connection speed and ability. Thus, multiple streams are actually available to the user, and may be seamlessly switched to if their connection drops lower or improves.

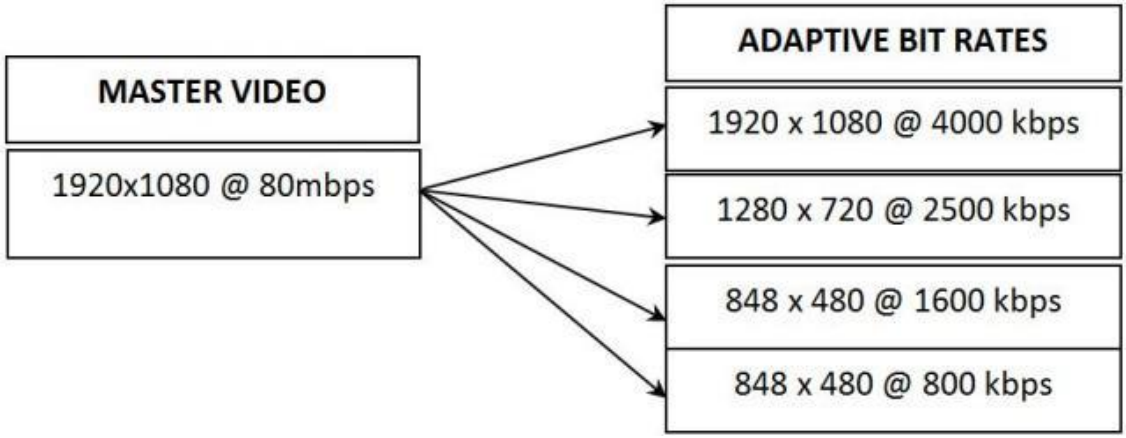

Figure 46. A master file that is scaled down to adaptive bit rates. Taken from (91).

An adaptive set is a package of transcodes for the same content (e.g. video) that span multiple bit rates and resolutions, and are meant to find a balance between connection speed and resolution. For stationary devices (e.g. set top boxes or computers), the frame rates, key frame intervals, audio sample rates, and so on should be the same within a set so that the player switches between bit rates as smooth and seamless as possible. For mobile devices, however, the connection/signal strength may fluctuate widely, thus the adaptive sets for mobile does not follow the adaptive sets for stationary devices. Here, "hard shifts" down in bit rates may be required to avoid mobile player crash and/or disengagement from content. I.e., since every device has different requirements for an ideal adaptive set, you need many versions of your content files to be playable across multiple devices. Depending on the target device and the context, the best transcoding practices vary a lot cross-device, which we can see from the charts in (91).

There are also other considerations that should be (and are) taken into account in relation to multi-device transcoding, which we will not discuss any further in this paper. These are:

- The transcoding method. Should the file be transcoded in constant or variable bit rates?
- Bitrate vs resolution considerations

- Resolution vs dimension considerations
- Buffer size.
- Profiles and levels

12.2.2 Content adaptation in a live migration solution

Live transcoding is transcoding of a stream encoded in one format to another stream encoded in another format, performed in real-time. This is what the proxy will have to do in a proxy-based live migration approach. There are two reasons why the transcoding should be done on the proxy, and not on the device itself:

1. We can't assure that the device actually supports the needed codec and/or format
2. Transcoding can be heavy on the CPU, and should at least be avoided on low-end devices (e.g. mobile phones)

Arguably, live transcoding will be most applicable in the following two types of applications:

- (Multimedia) streaming applications
- Video conferencing/chat applications

These are applications that deal with audio/video streams which may need to be converted following a migration.

12.2.2.1 Requirement considerations

The proxy/content adaptor will require a lot of context information

For each application, the content adapter entity (collocated with the proxy) needs to know what formats (and codecs) that are used, and which formats are associated with which devices. Additionally, it has to keep track of the currently active application sessions, and the devices associated with each session. Thus, upon a migration of a session, the content adapter needs to be informed about these parameters, and uses this information to decide the appropriate transcoding that has to be done. Thus, after a migration, the appropriate format transcoding of the session data will be performed, in real-time.

Such a content adapter will need to be able to work with various formats, and to convert between these. As previously described, this will require the content adapter to have a codec library, with several codecs that together, hopefully, supports all the various formats. It will also need a mux and demux library, with several multiplexers and demultiplexers capable of multiplexing and demultiplexing the various combinations of streams in container families.

The proxy may have to implement server-like functionality

Remember for instance Netflix's specific formatting and delivery engines, specifically adapted to the device it is serving. That way, content will be optimized for each requesting device. But if we introduce a proxy entity in this case, Netflix will continue to send optimized content to the device it *thinks* it is serving, since the server isn't aware of any migration. Thus, following a migration, the proxy may have to convert the content, ensuring it is compatible with the new device's capabilities. Thus, the proxy may have to implement *server-like functionality* (in this case it has to implement Netflix-like formatting and delivery functionality) if it is to provide clients with the adaptable/scalable features originally provided by the application server.

Strict time constraints leads to strict performance requests, and potential scaling issue

As mentioned, the proxy must be able to perform content adaptation in *real-time* (live transcoding) for the solution to be viable. Compared to streaming applications, real-time/interactive applications have stricter time constraints, as e.g. buffering can't be allowed in a real-time environment. If the latency of an RTC-app exceeds only 200 ms, the users will experience a loss of fidelity (92). If the latency exceeds 500 ms, it will result in an annoying, and possibly a destructive delay. Hence, in order for our solution to be tolerable, the total delay should not exceed 500 ms.

When using a live migration/proxy approach, we have to add the time it will take for the proxy to

- Process the packet (and possibly decrypt the media stream)
- Demultiplex, transcode and multiplex the media stream (in accordance with the target device/application's capabilities)
- Properly encapsulate (and encrypt) the packet before transmitting it

We see that encryption and decryption is mentioned in the points above. Many videoconferencing applications encrypt the audio/video stream to protect against potential eavesdroppers. Consequently, if the proxy has to do transcoding of such an encrypted media stream, it first has to decrypt it. And to ensure real end-to-end security, the proxy has to encrypt the converted media stream before sending it to the device. This adds up the total amount of work the proxy have to do in real-time, which puts even higher demands on its performance capabilities.

The time it will take for the proxy to do transcoding is obviously dependent on the proxy's CPU, workload etc. Here we also have to consider a potential scaling issue; the more users the proxy is serving, the more potential simultaneous live transcoding is needed. It's easy to see that this can put very high, possibly infeasible, performance requirements on the proxy.

12.2.3 Conclusion

Even though there is a trend towards similar formats and codecs across devices/platforms, we can't assume there will be a complete overlap, at least not in the near future. As the other endpoint is not made aware of a migration, the session parameters negotiated for the original session are maintained even after a migration. Thus, we may have to perform real-time content adaptation of both incoming and outgoing data following a migration. Here, we have identified several potential parameters the proxy will need to take into account, e.g.:

- The format (file and container) and codec
- Profiles and levels
- Application layer protocols (e.g. RTSP/RTP vs. HTTP)
- Encryption

The proxy is also responsible for gathering the needed context information about its connected devices and applications, as well as their capabilities, which may result in a substantial overhead.

Depending on the context, the proxy may have to perform adaptation (potentially by implementing server-like functionality) in one or several of the above-mentioned parameters, in real-time, which we argue may put infeasible high demands on the performance of the proxy, at least when considering a proxy that is to serve multiple concurrent sessions.

Even though such a live adaptation may be theoretically possible (e.g. by utilizing cloud technology and a cross-platform multimedia handler such as FFmpeg (93), we argue it can affect, or damage, the quality of the content. Consider e.g. a stream suited to a small-screen device, connected to a cellular network. If this stream is to be converted to fit a large-screen device, residing in a high-speed network, it will still be limited to the best bit rate, resolution etc. available in the original stream context. Hence, the stream, although adapted to conform to the new entities involved, will not necessarily be optimally suited to its new context.

Thus, we don't find a proxy-based live migration approach, dependent on a content adaption part, to be a desirable solution.

12.3 Proof of concept documentation

12.3.1 WebRTC interoperability notes

The WebRTC API is not yet finalized, i.e. it is still being drafted by the W3C WebRTC Working Group. As such, there is not yet complete interoperability between the browsers implementing WebRTC, which is why we only implemented the proof to work with Chrome. Fortunately, the WebRTC project keeps developers updated with interoperability news via their website, at: http://www.webrtc.org/interop.

We strongly recommend visiting this page should you be interested in developing an interoperable WebRTC-service (at least before it's finalized).

Interop notes

Currently, only Firefox Nightly (as of 1/30/13) and Chrome M25 Beta and later are interoperable, but require a small degree of adaptation on the part of the calling site. This Chrome/Firefox-interoperability, however, only refers to the MediaStream and the RTCPeerConnection APIs, not the RTCDataChannel API, as the creators of the PeerJS library have announced: "*DataChannel is available today in Chrome stable and Firefox Nightly. WebRTC DataChannels are not interoperable between Chrome and Firefox"* (94).

The current API-differences are:

W3C Standard	Chrome	Firefox
getUserMedia	webkitGetUserMedia	mozGetUserMedia
RTCPeerConnection	webkitRTCPeerConnection	mozRTCPeerConnection
RTCSessionDescription	RTCSessionDescription	mozRTCSessionDescription

Table 7. WebRTC API differences

"*Firefox and Chrome both prefix their interfaces and are likely to continue to do so until the standard is more finalized*" (79).

The WebRTC interop page also report other constraints/configuration issues one need to take into account, such as:

- Creation of local media streams
  - Chrome: element.src = webkitURL.createObjectURL(stream);
  - Firefox: element.mozSrcObject = stream;
    OR element.src = URL.createObjectURL(stream);

- DTLS-SRTP
    - "*Chrome does not yet do DTLS-SRTP by default whereas Firefox only does DTLS-SRTP. In order to get interop, you must supply Chrome with a PC constructor constraint to enable DTLS*"

Finally, Firefox offers a data channel on every offer by default (this is a stopgap till the data channel APIs are complete). Chrome mishandles the data channel m-line. In order to suppress the Firefox data channel offer, you need to supply a mandatory constraint to Firefox on CreateOffer. E.g.,

{'mandatory': {'MozDontOfferDataChannel':true}}.

However, in our solution this is not that relevant, as we require a datachannel.

WebRTC polyfill

Once WebRTC is finalized, we think it's safe to assume WebRTC will be interoperable between all major browsers. At that time, there may not be that much configuration needed to make it interoperable, though it's likely there still will be some platform-specific differences. Fortunately, WebRTC provides a polyfill to help insulate from these differences. It "*takes care of all these issues and lets developers write to the unprefixed W3C standard names"*.

The polyfill library can be found here:

https://code.google.com/p/webrtc/source/browse/trunk/samples/js/base/adapter.js

12.3.2 Deployment

The Proof of Concept index page can be reached at:

http://b2g.tele.no/

Telenor hosted our proof of concept, both the server, as well as the demo applications. Specifically, we were provided with an Ubuntu server hosted in a Virtual Machine at Telenor's data center, with the domain name b2g.tele.no and the IP address 193.156.17.80.

To host our platform, we used an Apache2 web server, to provide a basic web-site as an index for our services, which are all web-based. We also installed Node and its modules *socketio* and *daemon* which were necessary to run our Node server scripts. At this point, it was possible to manually login to our Ubuntu server and run the Node server scripts to start-up our services.

We also needed an automation mechanism that would start our servers automatically when the Virtual Machine started up, as well as some sort of periodical error control to make sure that the servers were restarted if an error occurred. In order to take care of that, we developed two more scripts. The first script is responsible for starting the Node servers (the migration server and the videochat server) and the second is responsible for stopping them (actually killing all processes related to "node"). Then we scheduled the execution of those scripts – first killing the node processes and then starting them – using the Cron System Daemon, so that it runs the scripts every day at a specific time. This way, if the servers encounter an error, they will be restarted.

12.3.3 Code documentation

You can inspect the full source code for all the generic scripts, both server- and client-side from here:

http://193.156.17.80/generic_parts/

The scripts are well documented throughout the code. Thus, if you find the need for a more thorough code documentation than we provide in this appendix, we refer to the above link.

12.3.3.1 The WebMS (migration_server.js)

The *migration_server.js* is the WebMS, a Socket.IO-JS running on Node. It is responsible for:

- Registering user's devices/applications to the migration server
- Keeping the connected devices/applications updated about the other connections
- Serve as a WebRTC signaling server for setting up WebRTC peer connections between the clients, used for session transfer and/or external view session establishment

The migration server is able to serve multiple applications, devices and users, and able to differentiate between these

By default, the transport mechanism is set to WebSocket in Socket.IO. Note that with Socket.IO we can set various fallback transport mechanisms if someone doesn't support WebSocket, by e.g. writing "io.set ('transports', ['websocket','flashsocket','htmlfile','xhr-polling','jsonp-polling']);"

Since establishing a WebSocket relies on the HTTP Upgrade mechanism, we needed to set up an HTTP server. Then we created the Socket.IO server, and instructed it to listen to the HTTP server we created.

The WebMS creates dynamic lists of currently connected clients and their properties in order to keep track of the associations. In our proof of concept, each application has its own WebMC, and thus creates its own socket connection to the WebMS. However, the WebMS is already implemented in a manner where the WebMC can represent multiple applications. Thus, each socket connection is associated with one user, but *can* be associated with multiple apps.

12.3.3.2 The WebMC (migration_signaling.js and migration_peerconnection.js)

As mentioned, the WebMC consists of two scripts; *migration_signaling.js* and *migration_peerconnection.js*. Below, we will present the content of the two scripts, and what they do.

The migration_signaling.js script

This is the client-side migration signaling script. It should:

- Connect the client to the WebMS automatically (once the user has provided user- and device name)
- Help establish an RTCPeerConnection session (and RTCDataChannel) between two clients by sending/receiving WebRTC offer/answer messages via the WebMS
    - When the client receives WebRTC-messages, we call the appropriate message handler in *migration_peerconnection.js*

It depends on the following methods in migration_peerconnection.js:

- handleMessage(from, data)

It depends on the following application specific methods/attributes (up to the application developer to implement):

- appId - a unique application identifier
- updateUser()
- updateConnections()

The migration_peerconnection.js script

This is the client-side peer connection script. It is responsible for:

- Creating an RTCPeerConnection between a source device and target device (peers)
- Creating a bidirectional RTCDataChannel on this peer connection for
    - state transfer and/or
    - an external view session

This script enables an application to do the following:

- Full migration. Transfer a running application from one device to another (session mobility). This can happen either
    - immediately, i.e. the application resumes immediately on the target device, or
    - paused, i.e. the application resumes in a paused mode on the target device
- External view. Establish an external view session between two devices, i.e. only transfer of the "view", where
    - The source device first transfers the state to the target device, which is then launched on the target device
    - The target device's view can be controlled from the source device
    - When completed, the user can choose to either:
        - Finish the migration, i.e. give control to the target device.
        - Exit the session, i.e. retrieve the view (state) back to the source device.

Depending on the application, it may be necessary to transfer the state over several packets. This script supports this by having the application developer specifying a boolean variable 'more'.

Upon full migration, this script:

- creates a peer connection (with a data channel) between the source and target
- transfers the state over the data channel (over one or several messages)
    - the state is transferred to the application-specific script to be launched
- closes the peer connection between the devices when the target device has received, acknowledged and launched upon the state

Upon external view, this script:

- creates a peer connection (with a data channel) between the source and target
- transfers the state over the data channel (over one or several messages)
    - the state is transferred to the application-specific script to be *externally launched*
    - the target device receives, acknowledges and launches (externally) upon the state

- the data channel is kept open for the source device to
  - send external view (e.g. playback) requests to the target device
  - finish migration to target device
  - exit the external view

The script depends on the following methods in migration_signaling.js:

- sendMessage(to, message)

The script depends on the following application specific methods/attributes (up to the application developer to implement):

- retrieveState() - returns a representation of the state of the application
- stateValid(state) - checks if state is valid, sets the state, and returns true if valid, false otherwise
- launch() - launches the application upon the state(s) received via stateValid
- exitExternalView - exits the external view (called after an 'exit external view' request)
  NB! Only needs to be implemented on an application that plans to use the external view feature
- takeControl() - makes it possible to control the application from this device (called after a 'finish' request in an external view session).
  NB! Only needs to be implemented on an application that plans to use the external view feature

12.3.3.3 Appspecific.js

"Appspecific.js"

Each application has to implement certain application-specific methods needed to make the application able to migrate.

Mandatory migration-specific variables to set:

- appId - a unique application identifier

Mandatory 'migration-specific functions' to be implemented in order to use the 'migration platform':

- updateUser() - should be used to e.g. display the user name and/or id to the user
- updateConnection() - should be used to e.g. display all the connected devices (to which the application session can be migrated) to the user.
- retrieveState() - called when the user wants to migrate from this device. Should retrieve the state of the current application session, to be sent to the target device.
- stateValid(state) - should be used to check if the given argument is a valid state. It should set the state and return true if valid, false if not valid.

- launch() - should be used to launch the state given the verified state argument. Should resume the app. session like it was on the other device
- stopApp() - stops the application session. This is called when a successful migration has completed from this device.

Optional functions to implement (NB! required if the application are to support external view):

- exitExternalView() - used by a controlling device to exit an external view (mostly GUI). Should return from an external view to a 'normal, local view'
- takeControl() - used by a controlled (external view) device after a completed migration to this device. Should enable controls and start a 'normal, local session' on this device.
- handleExtViewRequest(request) - used by a controlled device to execute the requested action, e.g. a playback action
- sendFinishRequest() - sends a request to finish the external view session, i.e. transfer the control to the target device currently being controlled. It should eventually call sendFinishExtViewRequest() in *migration_peerconnection.js*, which sends the request over the data channel.
- sendExitRequest() - sends a request to exit the external view session, i.e. transfer back the view to the source device. It should eventually call sendExitExtViewRequest() in *migration_peerconnection.js*, which sends the request over the data channel
- send<X>Request - sends an external view request, “X”, in an external view session, e.g. a playback request to the target device currently being controlled. It should eventually call sendExtViewRequest() in *migration_peerconnection.js*, which sends the request over the data channel.

Optional variables to set:

- more - a boolean variable used by migration_peerconnection
  - default: false - entails that the entire state will be sent over one message (in the data channel). This means that the retrieveState() method only will be called one time.
  - if true - the application has to send the state over multiple packets. This means that the retrieveState() method will be called several times until the whole state is sent. Then, 'more' must be set to false.

Recommended variables to implement:

- state - recommended to be used to set the state in stateValid() (if the state is valid), and then use this variable in the launch() method. However, if the application developer wants to implement this logic in another way, he is free to do so.

Recommended functions to implement:

- startMigration(remoteId,imm,extView) - a function triggered by the user to start a migration of type (imm, extView) to the target device identified by 'remoteId'. This is recommended to be used to e.g. change the GUI of the application. It should eventually call the method createConnection(id, imm,extView) in *migration_peerconnection.js* that actually starts the migration.

NB! In the case of RTC (real-time communication) apps:

In order to ensure a seamless and user-friendly migration in an RTC-app, e.g. a videochat app, we recommend to implement the following functionality upon a migration trigger:

- a notify()-function that sends a notification from the migrating user to the non-migrating user (via the app. server or the app. peer connection)
- upon this notification, the non-migrating client executes a wait()-function, waiting for a new call, and notifies the user about the situation

Example scenario:

- Assume two clients/devices, A1 and B1 (user A and user B), are in a RTC-session using application X. The app has implemented the migration client, and user A now wants to migrate to device A2.
- Before migrating, application X sends a notification (directly or via the app signaling server) from A1 to B1 to notify about the migration. Hence, B1 is made aware of the migration, communicates this to user B, and waits for a new call from user A (now from device A2).
- Simultaneously, device A1 and A2 uses the migration client to transfer the state (i.e. the address of B1), resulting in A2 calling B1.
- The session is continued between user A and B on devices A2 and B1, respectively, keeping the users well-informed throughout the migration.

The appspecific script depends on the following methods in migration_peerconnection.js:

- createConnection(remoteId, immediate, extView) - called to start a migration of type (immediate,extView) to 'remoteId'

..if external view is to be supported (see above):

- sendExtViewRequest(request)
- sendFinishExtViewRequest()
- sendExitExtViewRequest()

It also depends on the following method in migration_signaling.js:

- connectoToMigrationServer(userName,deviceName) - connects to the migration server, enabling this application to advertise itself, discover other devices, and migrate.

Additionally, our implementation utilizes jQuery(http://jquery.com/). Thus it is required that the applications also include JQuery.

12.3.3.4 RTCDataChannel comment

When we started testing our Chrome extension, we noticed that upon using the data channel to transfer the required data, some of the packets were dropped and the data channel was closed before the transfer was complete. The documentation of WebRTC is very limited as of today, especially when it comes to RTCDataChannel, as it is still under development. What we managed to find about this issue, though, came from discussions amongst other developers at various *developers.google.com* forums. Many appear to have run on the same issue and what appeared to solve it, was limiting the data throughput of the DataChannel. After finding out about this, we adjusted our extension to use an algorithm for sending, that tries to not exceed the sending rate of 3kbps. This has solved our issue of dropping packets and closing the channel. Unfortunately, we cannot base this choice on official documentation, as there is currently none on the subject.

Hopefully, official documentation regarding data channel size and throughput limits will be published in the future. It's not unlikely WebRTC will provide a built-in data transmission control functionality, eliminating the need for us implementing it.

12.3.4 Diagrams

12.3.4.1 WebMC state machine

When establishing an RTCPeerConnection, the clients (the WebMCs) pass through a variety of different states. Since our implementation is created in such a way that the migration should be automatically handled by the WebMC, we need to differentiate between the initiator (i.e. the source) and the target as well, as they shall act differently. The source should transfer the state while the target should receive it and launch upon it. We also need to be aware of the type of migration, as a *full migration* should result in closing the peer connection, while an *external view* migration should keep the peer connection open.

Figure 47 and Figure 48 show the state machine of the WebMC from an idle mode (i.e. being logged in to the WebMS, but not in any migration session) and during a migration

procedure. As we can see, it differentiates between source and target, as well as full and external view migrations.

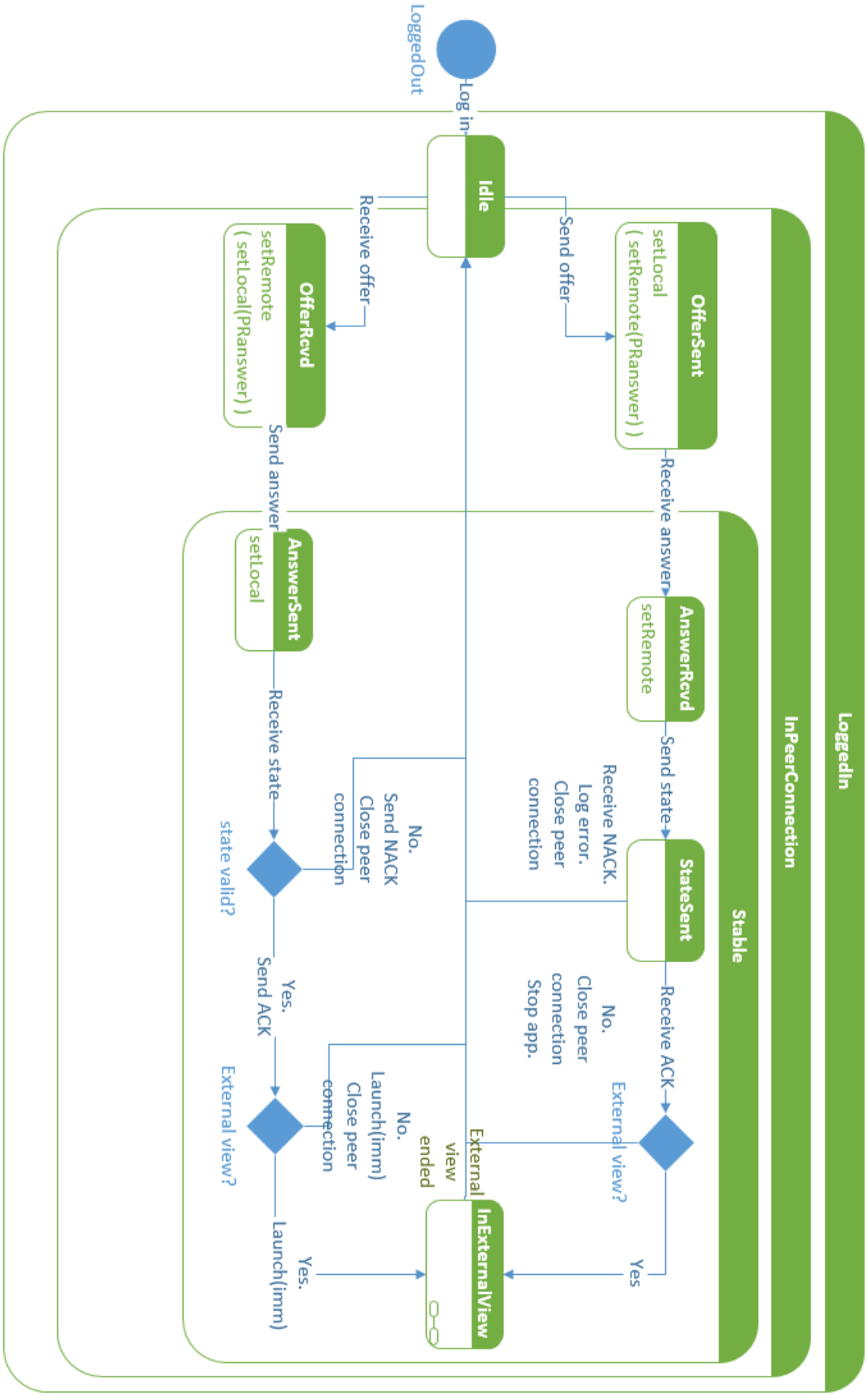

Figure 47. WebMC state machine, heavily based on the RTCPeerState Enum, specified in the WebRTC draft, at http://dev.w3.org/2011/webrtc/editor/webrtc.html#rtcpeerstate-enum

In Figure 47, the inExternalView state is a submachine state, i.e. a composite state whose internal details are not visible. Figure 48 shows the internal states of inExternalView.

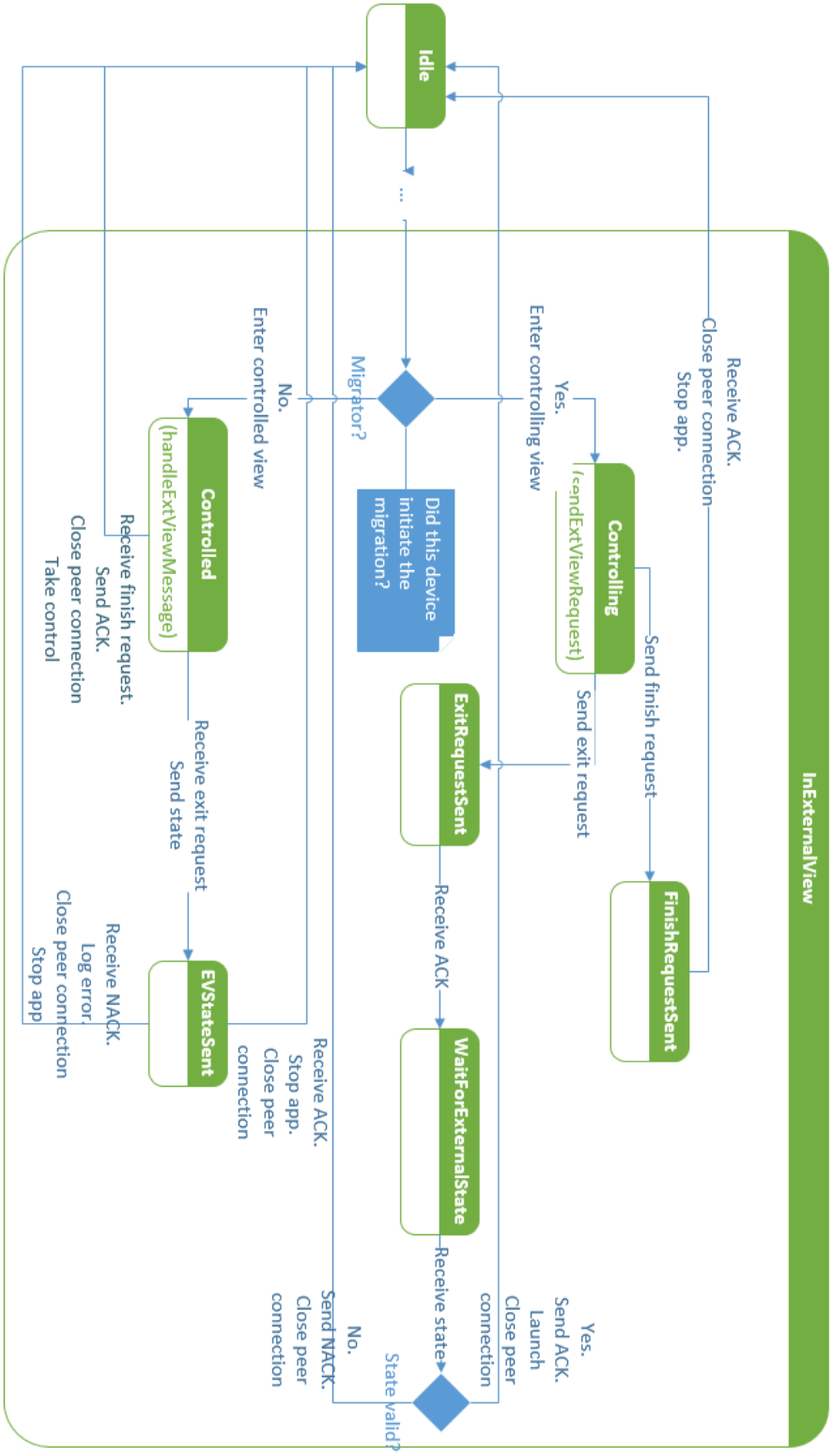

Figure 48. The inExternalView submachine state.

12.3.4.2 Full migration sequence diagram

In full migration call flow diagram presented in section 8.4.2, we simplified the following two subparts, which we will display and explain in more detail here:

1. WebRTC Peer Connection Setup
2. WebRTC Peer Connection Migration

WebRTCPeerConnection Setup

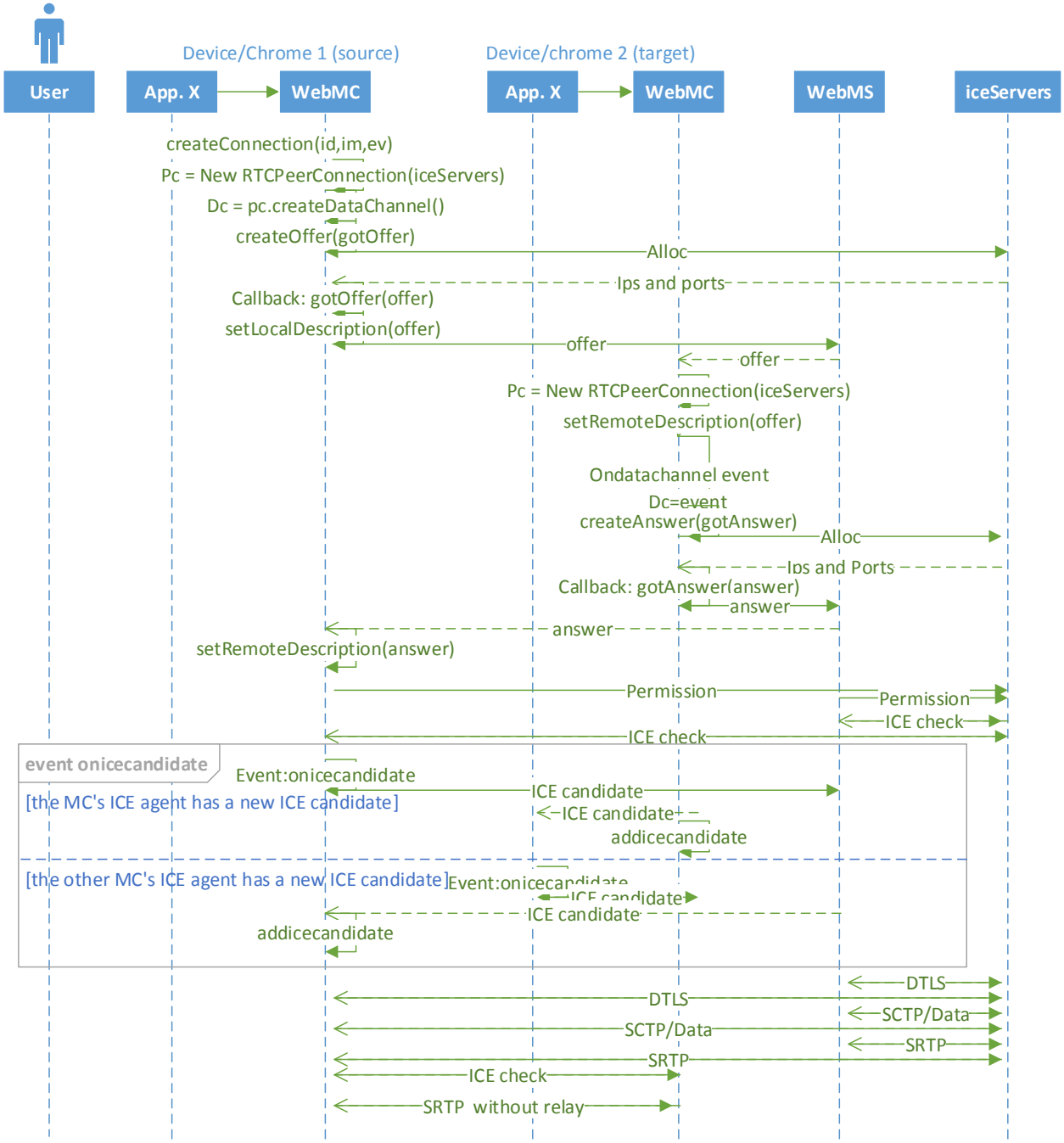

Figure 49. WebRTCPeerConnection Setup

Figure 49 shows how the two devices (i.e. the WebMCs) set up the peer connection using the WebRTC API. It is based on the instructions in the current (as of this writing) WebRTC API draft. Assume the user already has requested a migration from source to target device.

First, the source device creates a new RTCPeerConnection (pc), passing the address of Google's STUN server (iceServers). I.e., in our case, it says that Google's STUN server should assist us in creating a direct path of communication between source and target browser/MC. Since we are to establish a data channel between the devices, the source MC creates a data channel (dc) on the pc (dc = pc.createDataChannel(…)) as well. The source MC then wants to create and send an offer to the target. This offer must of course contain the request to open a data channel between the peers, but also the means of establishing that direct data path, i.e. one or several ICE candidate(s). That's why the MC sends an "Alloc" request to the iceServers, which responds with IPs and ports, i.e. ICE candidate(s), representing the alternative ways the target can reach the source. The source device sets the offer as its local description and sends it to the target, via the signaling server (since no direct path has yet been established).

When the target receives this offer (sent via the signaling server), it creates its own, local RTCPeerConnection (also passing the address of iceServers), and sets the remote description to be the offer. Since the offer contains a request to establish a data channel, an "ondatachannel" event also triggers, where the data channel is set to be that event. Similar to the source, the target does an "Alloc" request to the iceServers, which respond(s) with IPs and port values (ICE candidates / transport addresses). This, as well as potential data channel media negotiation parameters is set as the target's local description and sent to the source device as an answer, via the signaling server.

Upon reception of this answer, the source device sets it remote description to be the answer. Now it's time to establish the direct path, if possible. This phase we denote the ICE interface.

ICE interface

Remember that each peer, or MC, has its own ICE agent, as previously explained. It is responsible of finding one or several candidate pairs (communicating with the iceServers) for each MediaStream track (in our case: the data channel track) that form a valid connection. If at least one connection has been found for each media stream track, the ICE setup has completed, and a direct (or relayed) path has been established.

This is done in an incremental fashion, where different transport protocols are tested, and so-called ICE checks are performed, first with the iceServers, then directly between the peers. Should the ICE agent gather new ICE candidates (fired by an event "onicecandidate"), these are immediately communicated to the remote peer, where it is added.

Hopefully, the ICE Agents (with help from the iceServers) are able to create a direct path between the devices. If not, a relay server may have to be used.

For more about ICE, we recommend reading the WebRTC API draft (65). There you can e.g. read more about the RTCIceServer types, the different RTCIce state definitions, and the RTCIceCandidate types and events.

WebRTCPeerConnection Migration

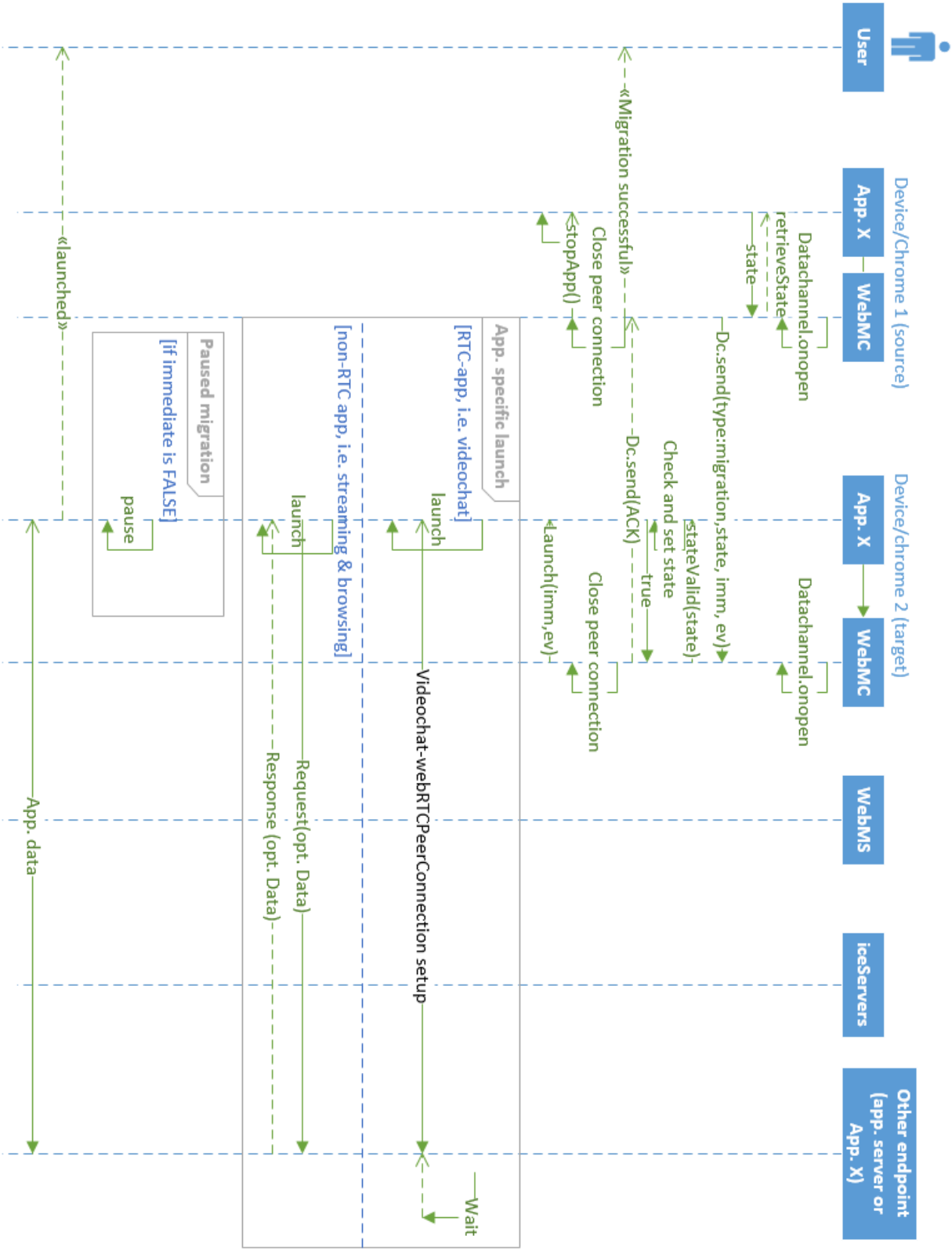

Figure 50. Proof of concept WebRTCPeerConnection migration

Figure 50 shows a migration from source to target device over the RTCDataChannel. As such, assume an RTCPeerConnection already has been established between the devices.

We created the migration by utilizing JavaScript's event-driven nature. I.e. the "onopen" event on the source device's data channel (dc) triggers the WebMC to invoke the application-specific retrieveState()-method, which returns the state of the application's session. The source then serializes this state object and sends it over the data channel, along with the type of message ("migration") and type of migration (imm/ev). On the target device, however, the "onopen" event doesn't trigger anything particular. This differentiating between source and target (or migrator and non-migrator) is determined by a Boolean value "migrator", which is false by default, but set to true if the createConnection() method of the WebMC is called.

Upon reception of the message, the target parses it, and, knowing it's a migration request with a state object, sends the state to the application to be set and validated. If the state is deemed valid, the WebMC sends an "ACK" message over the data channel to confirm a successful migration. The target WebMC then invokes the application-specific launch(imm,ev)-method, which launches the application upon the state argument in a fashion as determined by the imm/ev parameters.

If the request was for a full migration (i.e. not external view), the reception of the "ACK" message will result in the WebMC to close the peer connection and stop the app (via pplication-specific stopApp()). The migration has now ended successfully. Similarly will the target WebMC close the peer connection if the request was for a full migration. The only thing that remains is to launch the application upon the received state. This launch is application-specific, and not provided by our solution. However, we have developed a few demo applications that do implement this launch-function. One of which is the videochat application, also built on top of WebRTC.

Videochat-WebRTCPeerConnection setup

One of the demo applications we created, is the videochat application, also using WebRTC to establish a videochat call between peers. In Figure 50, we write "Videochat-WebRTCPeerConnection setup" for launching in the videochat case. Since our demo application is implemented in WebRTC, this setup is pretty similar to the WebRTC setup we've presented, but it also has to set up a media path for the local streams (i.e. the media captured from the web camera and/or microphone). As such, there are additional things happening in the videochat peer connection setup compared with the migration peer connection setup.

Section 10.4 in the current WebRTC Editor's draft (65) includes an example call flow between two browsers establishing a peer connection with media streams. As such, we regard it as a good example of the WebRTC setup between the two peers in our videochat demo application, though not entirely complete: "*This does not show the procedure to get access to local media or every callback that gets fired but instead tries to reduce it down to only show the key events and messages.*". However: "*This example flow needs to be discussed on the list and is likely wrong in many ways."*

The diagram can be found here:
http://dev.w3.org/2011/webrtc/editor/webrtc.html#simple-peer-to-peer-example
(section 10.4 – Call Flow Browser to Browser). Or you can access the direct link to the image here:

http://dev.w3.org/2011/webrtc/editor/images/ladder-2party-simple.svg

12.4 A how-to-demo tutorial

In this section we will provide you with simple instructions on how you can test each demo application, and thus get a feel of the behavior of our implemented proof of concept.

You can reach the index page of our proof of concept here:

http://b2g.tele.no/ or http://193.156.17.80

Before you're demoing, please make sure you fulfill the following device/platform prerequisite:

Supported devices/platforms: Chrome's stable versions on both PC and mobile (Android) versions. To be safe, make sure you have the latest Chrome version, and ensure WebRTC is enabled by visiting this URL: chrome://flags/

Note: If you're experiencing problems on your mobile device, we urge you to test it on PC only, as we unfortunately haven't had the necessary equipment to do any testing on the mobile platforms.

12.4.1 HTML5 video streaming

Immediate/paused migration

1. Go to http://193.156.17.80/streaming_HTML5/ on two devices (or open two tabs in the same browser window).

2. Push the "Show migration GUI"-button. Log in with the same username on both devices, but with different device names, e.g. "device1" and "device2" (or any two different names).

3. On "device 1": Play any video (from the drop-down list or by providing an URL to a HTML5-supported video).

4. On "device 1": Next to "device2" in the list of available devices in the migration GUI, push the "(Immediate migration!)" to migrate immediately to "device2".

The video playing session should now have been migrated to "device2" immediately, i.e. the video should continue playing on "device2".

5. On "device 2": Next to "device 1" in the list of available devices in the migration GUI, push the "(Paused migration!)" to migrate paused to "device1".

The video playing session should now have been migrated to "device 1" in a paused mode, i.e. the video should start in a paused mode at the time it was stopped at "device 2".

External view

1. Go to http://193.156.17.80/streaming_HTML5/ on two devices (or open two tabs in the same browser window).
2. Push the “Show migration GUI”-button. Log in with the same username on both devices, but with different device names, e.g. "device1" and "device2" (or any two different names).
3. On “device 1”: Play any video (from the drop-down list or by providing an URL to a HTML5-supported video).
4. On “device 1”: Next to “device2” in the list of available devices in the migration GUI, push the “(Immediate external view!)” OR the “(Paused external view!)” button to perform an immediate OR paused external view migration to “device2”.

The video playing session should now have been migrated to “device2” immediately OR paused, i.e. the video should be loaded on “device2”, either resuming playback immediately OR in a paused mode. Additionally should “device 1” display “External controls”, and “device 2” should only display the video player, i.e. no controls.

5. Perform any amount of playback actions of your choosing to control the external video playback, e.g. play, pause, mute, unmute, video seek, load another video etc.
6. Finish or exit
    a. Finish: Push the “Finish” button to complete the migration to “device 2”.
    b. Exit: Push the “Exit” button to exit the external view session, i.e. retrieve the external video playback on “device 2” to “device 1”.

The video playing session should now have been either fully migrated to “device2”, or retrieved at “device 1”. Both devices should display a “normal view”, in the sense that there are no ongoing peer connection and/or external view session between the devices.

Note on use:

The HTML5 video streaming demo application can be used in two “modes”:

1. *Public.* By logging in with the migration username “public”, the user can see any other user logged in as “public” as well, and is able to send a video to any other user logged in as “public”, which kind of makes the application a “public video sharing” application.
2. *Private.* By logging in with another migration username, e.g. “private”, the device will only be visible by other devices logged in as “private” as well, which makes it a private HTML5 video migration application.

12.4.2 YouTube video streaming

Immediate/paused migration

1. Go to http://193.156.17.80/streaming_youtube/ on two devices (or open two tabs in the same browser window).
2. Push the “Show migration GUI”-button. Log in with the same username on both devices, but with different device names, e.g. "device1" and "device2" (or any two different names).
3. On “device 1”: Play any video (from the drop-down list or by providing an URL to a HTML5-supported video).
4. On “device 1”: Next to “device2” in the list of available devices in the migration GUI, push the “(Immediate migration!)” to migrate immediately to “device2”.

The video playing session should now have been migrated to “device2” immediately, i.e. the video should continue playing on “device2”.

5. On “device 2”: Next to “device 1” in the list of available devices in the migration GUI, push the “(Paused migration!)” to migrate paused to “device1”.

The video playing session should now have been migrated to “device 1” in a paused mode, i.e. the video should start in a paused mode at the time it was stopped at “device 2”.

External view

1. Go to http://193.156.17.80/streaming_youtube/ on two devices (or open two tabs in the same browser window).
2. Push the “Show migration GUI”-button. Log in with the same username on both devices, but with different device names, e.g. "device1" and "device2" (or any two different names).
3. On “device 1”: Play any video (from the drop-down list or by providing an URL to a HTML5-supported video).
4. On “device 1”: Next to “device2” in the list of available devices in the migration GUI, push the “(Immediate external view!)” OR the “(Paused external view!)” button to perform an immediate OR paused external view migration to “device2”.

The video playing session should now have been migrated to “device2” immediately OR paused, i.e. the video should be loaded on “device2”, either resuming playback immediately OR in a paused mode. Additionally should “device 1” display “External controls”, and “device 2” should only display the video player, i.e. no controls.

5. Perform any amount of playback actions of your choosing to control the external video playback, e.g. play, pause, mute, unmute, video seek, load another video etc.
6. Finish or exit
    a. Finish: Push the “Finish” button to complete the migration to “device 2”.

b. Exit: Push the “Exit” button to exit the external view session, i.e. retrieve the external video playback on “device 2” to “device 1”.

The video playing session should now have been either fully migrated to “device2”, or retrieved at “device 1”. Both devices should display a “normal view”, in the sense that there are no ongoing peer connection and/or external view session between the devices.

Note on use:

The YouTube video streaming demo application can be used in two “modes”:

1. *Public.* By logging in with the migration username “public”, the user can see any other user logged in as “public” as well, and is able to send a video to any other user logged in as “public”, which kind of makes the application a “public video sharing” application.
2. *Private.* By logging in with another migration username, e.g. “private”, the device will only be visible by other devices logged in as “private” as well, which makes it a private video migration application.

12.4.3 Videochat

Videochat migration

1. Go to http://193.156.17.80/videochat/ on three devices (or open three tabs in the same browser window).
2. ALLOW(!) access to web camera and microphone when prompted to do so.
3. Log in as migration AND videochat user: "A" on two devices/tabs, and “B" on the third, with device names “A1”, “A2” and “B1”, respectively.

Now, the “A “videochat peers should be able to see one “B” videochat peer, and the “B” should be able to see two “A” videochat peers.

4. From “A1”: Call “B”

Now, user “A“, at device ”A1” should be in a videochat session with user “B” at device “B1”.

5. Write something in the private chat from both the users in the active videochat session.
6. From “A1”: Push the “Show migration GUI”-button, and push the “(MIGRATE TO THIS)” hyperlink next to “A2” in the list of available devices.

The videochat call AND private chat should now be migrated to device “A2”. During this migration, user “B” should be displayed a message “migrating….please hold” until device “A2” calls “B1” and resumes the videochat session.

7. Hang up from any device.

Note on use:

The videochat application can be used in two “modes”:

1. *Public.* By logging in with the migration username “public”, the user can see any other user logged in as “public” from the migration GUI as well. Thus, he/she is able to transfer a videochat call/session to any other migration user logged in as “public”, which kind of makes the application a “public call transfer” application. In this case, not very usable, but still showcasing an implementation of call transfer in WebRTC!
2. *Private.* By logging in with another migration username, e.g. “private”, the device will only be visible by other devices logged in as “private” as well, which makes it a private videochat migration application.

12.4.4 Browsing

Before you can test the browsing extension (i.e. migrate a browsing session from one device to another), you need to download and install the Chrome extension. To do this, follow the following setup instructions:

Setup:

1. You need two (2) computers with Chrome installed

On both computers:

2. Download the extension (zip-file) at http://193.156.17.80/browsing_extension/extFinal.rar
3. Unzip the file. You will get a folder named "extFinal"
4. Open Chrome
5. Go to chrome://extensions/
6. Check "Developer mode"
7. Click "Load unpacked extension..."
8. Browse to and choose the "extFinal" folder you just downloaded.

The extension should now be accessible as an extension/icon (see Figure 51) on the top right part (to the right of the URL input field).

Figure 51. The browser extension icon

Browsing migration

You're now ready to test the browsing extension. To perform a migration of a Chrome browsing session, do the following:

1. Open Chrome on both devices.
2. From your chosen "source" device: open a few tabs of your choosing (these will be transferred to the target device upon migration).
3. Open the extension (push the extension symbol located in the top right of the browser window) on both devices.
4. Log in with the same migration username, but different device names, e.g. "source" and "target".

You should now see the other device appear in the list of available devices.

5. From "source" device: Choose to migrate "immediate" to the other device.

Watch as the "target" device loads the same tabs that are open in the source device. In the background (visible from the console log – CTRL+SHIFT?+i), the "source" device also receives and stores cookies and history files.

NB! Note that the browsing extension only works in a PC environment, as Chrome apps and extension currently are not supported on mobile. Google has said that they have no plans to announce extensions on mobile at this time (95).