Middleware Mobility Services for Self-adaptive Multimedia Processing in Ubiquitous Computing Environments

Francisco Javier Velázquez-García

April 30, 2019

Thesis submitted for the degree of Philosophiae Doctor

Abstract

The introduction of mobile computing devices made it possible for users to bring their running applications into different physical environments, sometimes surrounded by other devices with different multimedia capabilities. However, despite the advances in distributed systems and mobile computing, the development of multimedia applications that can use a changing set of heterogeneous devices seamlessly, continues to be very difficult. We identify two main reasons for this difficulty. First, the inherent unpredictability of changes in the user environment, the application runtime environment, and network conditions. Second, typical inter-process communication (IPC) mechanisms, such as the Berkeley Unix sockets or the standardized Portable Operating System Interface (POSIX) shared memory, are not designed for mobile applications.

For users to take advantage of the changing set of surrounding devices, the applications (or parts of them) should be present in each device. However, pre-installing and configuring all applications in all devices the users might want to use, results impractical, especially if the receiving devices change frequently. Alternatively, seamless fine-grained application mobility can enable users to move their applications on demand, as the users encounter new devices.

This thesis presents research to ease the development of multimedia applications that can move to different devices in a fine-grained and seamless manner, while preserving multimedia sessions that produce or consume multimedia streams, as in video conferencing applications. We aim to develop components for a middleware that is based on an autonomic adaptation loop, and a framework that offers an Application Program Interface (API), which embodies an abstract design for mobile multimedia applications adhering to the ubiquitous computing paradigm.

The middleware is designed to use multi-dimensional utility functions that allow users and developers of multimedia components to do a preference elicitation, and select the multimedia pipeline configuration that fulfills the preference of the user in a given context. The architectural constraints applied to modeled pipelines as graphs mitigate the combinatorial explosion when autonomously creating the variability search space of pipelines. This approach enables multimedia applications to apply adaptation techniques unforeseen at design time.

We also provide an API for two IPC mechanisms with mobility services. The first service allows Berkeley Unix sockets to migrate between devices; this service uses commodity hardware, e.g. a smartphone, as a proxy to hide the mobility of endpoints from legacy applications. The second service shares the data needed by distributed components among collaborative devices; this service aggregates the available CPU and bandwidth resources by implementing a publish-subscribe (PUB/SUB) mechanism that enables devices to share the data that they are already consuming.

The proposed services are evaluated by applying analytical and experimental research methods including testing of prototypes. In conclusion, the resulting services at middleware-level are a step forward to detach multimedia applications from the host device at run time, and move them to heterogeneous devices in a seamless and transparent manner. Further research in this direction will enable users to take full advantage of the constantly changing set of multimedia-capable devices that surround them.

Acknowledgments

I am grateful with many people and organizations that have made this PhD thesis possible. Everyone here named helped me in one way or another. I would like to thank my family, for being supportive of all what I want to do. Specially motivating is my father's unbreakable willpower for education and knowledge, as the foundation for progress in his life, despite all the adversities he faced during his school time in the poor countryside of Mexico, and while continuing his education in Mexico City without any financial or moral support from his family.

I would like to thank my supervisors, Prof. Frank Eliassen, Dr. Håkon Stensland Kvale, and Prof. Pål Halvorsen for all their mentorship, advice and feedback. The advice and support from them made possible the continuation and completion of this thesis. I would like to thank the master students I co-supervised: Håvard Andersen Stigen, Haakon Wilhelm Ravik, Goran Karabeg, Andič, and Tomas Gryczon. They helped me to improve my understanding of the topics we discussed.

I would like to thank Dr. Kristoffer Robin Stokke for helping me to see alternatives when I was stuck. My stay in Simula Research Laboratory was short, but I am very thankful to everybody in the CASPER research group for sharing their knowledge, working culture, and motivating me to finish this dissertation. Especial thanks to Dr. David Hayes and PhD candidate Jørgen Dokken for the fruitful discussions on mathematics, Dr. Petra Filkukova, and master students Hugo Wallenburg, and Asad Sajjad Ahmed. I am also grateful with Lena Zhuk for giving me all the needed emotional and moral support during the most difficult years of my PhD.

I would like to thank the GStreamer community for all what I have learned from them about multimedia processing and values of Free Libre Open Source Software (FLOSS), and the GStreamer Foundation for financing my visit to the hack festival in Sweden on the Spring of 2018. I am grateful with Rubén Darío Romero y Cordero Gavilanes for introducing me to Varnish Software, and being a supportive colleague and friend for many years. Varnish Software employed me in a part-time position that provided me the flexibility and financial support after my research funding finished. I am thankful to my colleagues in Varnish Software for constantly motivating me to finish this dissertation. Discussions on mathematics with Dr. Pål Hermunn Johansen were of great help, as well as the agreements with Espen Braastad on flexibility of working hours. I also would like to thank my current employer Make.TV for the time flexibility to prepare the defense of my thesis.

I would also like to express my gratitude to Prof. Thomas Plagemann and Dr. Ellen Munthe-Kaas for giving me the opportunity to become a PhD student. From the DMMS research group, I would like to thank my colleagues Fabrice Starks for the many hours of philosophical discussions on what means to be a PhD student, Stian Sjøli, Hans V. Hansen, Piotr Strebny, Piotr Kamisińkski, and Azadeh Abdolrazaghi. I also would like to thank Vasileios Mavroeidis from the PSE research group for the many interesting discussions and for borrowing his desk for months.

Contents

List of Figures

List of Tables

Glossary

application mobility mobility type where users can move their running applications across multiple heterogeneous devices in a seamless manner. i, xv, xix, 17–19, 41, 42, 44, 62, 75, 87, 105–107, 117

architectural constraint design knowledge from the application developer with the purpose to reduce combinatorial growth by limiting configuration variability. i, 78, 108–110

calm computing approach to ubiquitous computing, where computing moves back and forth between the center and periphery of the user's attention [198]. 18

compositional variability alternatives in components ($v \in V$), input ($i \in v.I$), and output ($o \in v.O$) connectors. 78, 108

content-based action of transforming content to adapt to device capabilities. 74, 77, 106

context any information that characterizes the user surroundings, preferences, application running environment, or network conditions, which impacts the functional and non-functional requirements of an application. i, xix, xx, xxiii, xxvi, 4, 6, 16–19, 36, 45–47, 61, 67

context-aware extensive and continuous use of any information that characterizes the user surroundings or application running environment, which impacts the processing of multimedia presentations. 4, 19, 76, 86, 107, 117

destination device host computer device where an application resumes execution after application mobility is done. 8, 102

device host computer where an application or process runs. xvi, xxiii, 3, 8

distributed shared memory data communication abstraction where memory segments are shared amongst a set of devices. 141

everyday device commodity devices including smartphones, desktop computers, tablets, and smart TVs. 107

fidelity application-specific output quality. 74, 77, 106

framework a set of services provided by an API that embodies an abstract design for solutions to a number of related problems in application mobility. i, 7, 9, 10, 17, 39, 44, 58, 63, 65

functional path in graph theory terms, path is an abstraction of the sequentially connected components that process a stream to do a certain task, e.g., capture video from a webcam and send it over the network. 11, 44, 45, 95

functional stage group of components by functionality. 11, 44, 45, 52, 79, 109

GStreamer open source multimedia framework that provides a library for constructing graphs of media-handling components. 11, 43, 45, 59, 75, 81, 83, 105–108, 112–115, 117, 118

GStreamer element basic building block for a media pipeline in the GStreamer multimedia framework, e.g. encoder or video sink. In this thesis, a GStreamer element is equivalent to a specific implementation of a multimedia pipeline component. 17

I/O input or output communication between a computer device and its users, other devices (via a network) or the outside world. The hardware used as interface to do this is called peripheral. 4, 8, 12, 16, 17, 27, 29–31, 33, 42, 52

JIT Just In Time compiler. 17

K knowledge created and used by the phases in the Monitor, Analyze, Plan, and Execute (MAPE) adaptation control loop. 50, 51, 58, 90

locality data that has been accessed recently has a *temporal locality*, data in a near memory address has a *spatial locality*. 8, 145

middleware software that mediates between an application program, and hosting operating system or a network, it manages the interaction across heterogeneous computing platforms. i, xviii, 4, 7, 10, 12, 15, 17, 19, 39, 44, 45, 51, 54, 56–58, 63, 65

mobile application applications whose processes can migrate between devices. i, 8, 11, 51, 59, 62

modality of a particular sense from the sensory system, as senses of sight, hearing, taste and touch. 4, 5, 16, 74, 75, 77, 106, 108–112, 114, 115, 117

multimedia any *collection* of data including text, graphics, images, video (moving images presented as a sequence of static images), audio, tactile modalities or any system for processing or interacting with such data. 26

multimedia application application that processes data in one or more distinct multimedia modalities. i, 3, 51, 87

multimedia content something (e.g. a person, object or scene) selected by, e.g. an artist, a photographer or multimedia developer, for multimedia representation . 26, 51, 66

multimedia pipeline sequentially connected components that process multimedia presentations. i, 42, 52, 66, 67, 75–77, 105–108, 117, 163, 164

multimedia presentation multimedia content composed by a collection of media. xvii, 7, 26, 31, 51, 65, 74, 75, 105–107, 117

NaN an IEEE floating point representation used to detect an unwanted pipeline variant. 47, 50

NP Nondeterministic Polynomial time, property of computational decision problems solvable by a nondeterministic Turing Machine in a number of steps that is a polynomial function of the size of the input (see [98] for full definition). xvii

NP-hard a property of computational search problems, solving an NP-hard problem in polynomial time would make it possible to solve all problems in class NP in polynomial time (see [98] for full definition). 11, 25, 42, 66, 85, 89

POSIX A set of IEEE standards designed to provide application portability between Unix variants. IEEE 1003.1 defines a Unix-like operating system interface. 12

paradigm an example, model or pattern containing the assumptions, ways of thinking, and methodology that are commonly accepted and shared by members of a discipline, group or scientific community. i, xix, 3, 4, 23, 39, 65, 73, 87, 105, 140

parameterization variability different configuration of pipeline components due to the properties of components themselves ($v.P$), properties of input ($i.P$) and output ($o.P$) connectors, and properties of modalities ($m.P$). 78, 108

path sequence of successive edges through a graph (where a vertex is never visited more than once); abstraction of the sequentially connected components to process one multimedia stream. 110

platform 1. Support software for a particular activity, as in "This program provides a platform for application mobility". 2. Specific combination of hardware, operating system or compiler, as in "this middleware manages the interaction between applications and heterogeneous computing platforms". xvii, 4, 21

preference elicitation process where the autonomic manager clearly defines the utility of a pipeline variant based on the preferences of the user and developer of pipeline component. i, 46

process migration technique whereby an active process is moved from one machine to another, while continuing normal execution and communication. 58

PUB/SUB publish-subscribe pattern originally proposed in [145]. ii, 12, 25, 56, 66

retargeting process of adapting an image or video from one screen resolution to another to fit different displays. 74, 77, 106

safety predicate every component in a multimedia pipeline is always able to process data in synchrony to their reference clock, and the current configuration provides a high enough utility to the user. 18, 19, 61

SeamCrop retargeting of videos which combines cropping and seam carving (content-aware image resizing). 29, 59

self-adaptive application that reacts to changes in the context by changing its safety predicate accordingly. 18, 41, 44, 51, 65

self-awareness application that is able to monitor and analyze its context. 18, 19, 42, 44, 65

self-configuration application that reacts to context changes, and change the connections or components of the application, to restore or improve the safety predicate. 18, 41, 44, 65

self-managing ability to make decisions w.r.t. context changes to maintain, improve or restore the safety predicate without human intervention. 44

self-optimization application that improves (maximize or minimize) the value of a predefined objective function. 18, 19, 41, 44, 65

service work performed or offered by an entity, such a server or a software library. ii, 3, 4, 15, 19, 20, 65

source device host computer device where an application executes before application mobility starts. 7, 8

stub a routine that does not need to contain any code, but it is only present to prevent errors when linking a program with a run-time library. 68

subgraph subgraph that represents one multimedia pipeline, $g \in G'$. 111

system the entire computer system, including I/O devices, the supervisor program or operating system and possibly other software. 6, 8, 9, 15, 16, 166, 167

ubiquitous computing paradigm where computing is made to appear anywhere and anytime, meaning that users, applications and devices are nomadic, and spontaneous interactions are a norm. i, 3, 7, 11–13, 18, 39, 62, 65, 66

utility degree to which a particular configuration variant has the potential to satisfy the user's needs. The value of a utility is a real number between zero (worse) and one (best). xx, 46

peripheral any part of a computer device other than the CPU or working memory, e.g., cameras, monitors, speakers, microphones, keyboards, joysticks, mice, disks, printers, scanners, to mention just a few [98]. xvi, 5, 15–17, 27, 29, 30, 33, 58

utility function mathematical relation such that each variation in the context of the application and the user is associated with a real number between zero and one. 46

Acronyms

API Application Program Interface. i, ii, xvi, 5, 7, 12, 17, 19, 38, 39, 44, 45, 51, 54–59, 62, 65, 67, 127, 137, 140, 142, 146, 150, 153, 156, 165

BRGC Binary Reflected Gray Code. 45, 80, 97, 110, 116

CORBA Common Object Request Broker Architecture. 33

DAMPAT Dynamic Adaptation of Multimedia Presentations in Application Mobility. 39, 44–46, 51–53, 60, 87, 88, 90, 104, 105, 107, 108, 110, 111, 113, 115–118

DSM Distributed Shared Memory. 33, 141–144, 158

DSPL Dynamic Software Product Line. 34, 50, 51, 76, 105, 107

ECA event-condition-action. 98, 103

GUI Graphical User Interface. 166

HTPC home theater PC. 3, 16, 18, 30

HW hardware. 150–152

I/O Input/Output. xvi, 4, 8, 12, 16, 17, 27, 29–31, 33, 42, 52, 74, 78, 89, 90, 92, 101, 106, 120, 155, 157

IBM International Business Machines. 23

IP Internet Protocol. 7, 54, 55, 120, 121, 124–131, 136, 137, 156, 158

IPC inter-process communication. i, ii, 12, 24, 36, 38, 54, 57, 62, 66, 150

JIT Just In Time. xvi, 17, 39, 58

JVM Java Virtual Machine. 33, 58

K Knowledge. xvi, 23, 39, 40, 50, 51, 58, 90

MAPE Monitor, Analyze, Plan, and Execute. xvi, 23, 39, 40, 42, 44, 50, 51, 58, 61, 63, 76, 90, 105, 107, 108

NaN Not-a-Number. xvii, 47, 50

NP nondeterministic polynomial type. xvii

PDA Personal Digital Assistant. 16

PIM Personal Information Management. 12, 24, 33, 36, 37, 41, 57, 60

POSIX Portable Operating System Interface. i, xviii, 12, 56, 67, 127, 137, 150

PUB/SUB publish-subscribe. ii, xviii, 12, 25, 56, 66

QoE Quality of Experience. 17, 18, 20

QoS Quality of Service. 17, 18, 68

SOCKMAN SOCKet Migration for multimediA applicatioNs. 32, 55, 67, 119, 121, 123, 131, 137, 159

TCP Transmission Control Protocol. 7, 11, 54, 57, 62, 66, 120, 124, 125, 128–130, 133–137, 150

TRAMP TRAMP Real-time Application Mobility Platform. 39, 57, 58, 120, 129, 153, 156, 157

UDP User Datagram Protocol. 7, 11, 54, 57, 62, 66, 120, 121, 125, 126, 128, 131, 133, 136, 137, 150, 158

VoD Video on Demand. 59

Symbols

α response time spent to create one pipeline with functional stages. 116

β time spent to filter vertices in functional stage. 115, 116

C1 first consumer of data. 151, 152

C2 second consumer of data. 151, 152

C3 third consumer of data. 151, 152

DL average data loss. 132

E set of edges that represents the connection or pipe between the output and input connectors of two pipeline components. 77, 94, 97, 108, 111

ε weight of property $p \in P$. This symbol is equivalent to we. Rank, priority and importance are synonyms of weight in this thesis. xxvi, 47, 48, 50

G multigraph that represents all possible multimedia pipelines in a host device. 77, 94, 108

Γ set of paths $\{w\}_i$ for the same modality. 110

g subgraph that represents one multimedia pipeline, $g \in G'$. xxiv, 45, 46, 97–99

G' set of subgraphs that represents one or many multimedia pipelines for a given context, $G' \in G$. xxiv, 45, 46, 97, 111

$g.P$ set of properties in a subgraph that represents a pipeline. 46

$g.p$ variable that contains value of property in subgraph. xxv, 47, 98, 111

H all weighted utility elements h. 47, 48

h weighted utility element τ. xxiv, 47, 48

$i.P$ parameterization variability of input connector i. 46, 78, 95

i index of element in a set, or input connector $i \in v.I$. xi, 109, 110

i input connector of component $(i \in v.I)$. xxiv

k number of paths for the same modality, defined by application developer. 101, 110

Λ multi-dimensional utility. 48, 49

υ multi-dimensional utility. 48–50

l_d direct data transfer from multimedia application to legacy application. 132

l_p data transfer from mobile multimedia application to legacy application through proxy. 132

M set of modalities $\{m\}_i$. 97, 108, 114

m modality of stream, e.g., audio, video or text, $m \in M$. 108, 110

$m.P$ parameterization variability of modality. 46, 78, 95

n number of independent variables that constitute the dimensional space of the overall utility function of a pipeline. 48, 49

n maximum index number of set of graphs $g \in G'$. 97

n cardinality of set of paths Γ for the same modality. 101, 110, 116

n network delay. 152

$\mathbb{R}$ set of the real numbers. 46, 48, 49

η time needed to create all paths $\{w\}_i$. 116

$\mathcal{O}$ order of computational complexity, number of steps or arithmetic operations, i.e., big O notation. 101, 110, 116

o application (middleware) overhead. 152

$o.P$ parameterization variability of output connector. 46, 78, 95

PL minimum number of packet loss. 132

P probability of additional packet loss. 132

P original producer or source of data. 151, 152

p property of multimedia pipeline. 46

P set of properties in a pipeline. 46, 80, 94, 97, 98

σ weighted utility function. 47

s functional stage. 109, 114, 115

Θ all functions θ to get the weighted utility. 47, 48

t total producer-to-consumer delay. 152

T all possible instantiation of utility elements. xxv, 46, 49

τ instantiation of utility element of T. xxiv, 46–48

θ function to get weighted utility. xxv, 47

t_r time to reinstate the socket state. 131

t_s time to export the socket state. 131

Υ overall weighted multi-dimensional utility. 48, 112

μ utility function. 46, 47

ut instantiation of one dimensional utility function that takes two arguments: $u.p$ and $g.p$. 47, 98, 111

u user variable that contains preferred values of properties p. 47, 111

$u.p$ variable that contains value of preferred property specified by the user, or measured value from user's physical environment. xxv, 47, 98, 111

$u.p.we$ priority of property ($u.p$) given by the user. 98, 111

V set of vertices v that represents components of a pipeline. 77, 94, 108, 115

$v.P$ parameterization variability of vertices. 46, 78, 95

v vertex that represents a pipeline component, $v \in V$. xv, xxiv, xxvi, 97

W set of paths. 45, 97, 110, 116

w sequence of connected components that process one stream, $w \in W$, i.e., functional path. 45

we weight of property $p \in P$. This symbol is equivalent to ε. Rank, priority and importance are synonyms of weight in this thesis. xxiii, 47, 98

X representation of all possible instantiated context x. 46, 47

x instantiation of context. 46

Part I

Overview

Chapter 1

Introduction

In this chapter, we establish the context of the thesis. Section 1.1 explains the motivation and background. Section 1.2 states the problem statement and the research questions. Section 1.3 describes the aim of the thesis and application domains targeted in this work. Section 1.4 outlines the research methods applied in this thesis, and research phases. Section 1.5 describes the contributions of this thesis and summarizes the conclusions based on results from research publications included in Part II. Section 1.6 describes the structure of this thesis.

1.1 Motivation and Background

Since the origins of electronic computers, there has been a constant development of multimedia-capable devices, multimedia applications, and Internet services. The introduction of mobile computing devices allowed users to bring their running applications on the move. Consequently, the users' environment, including the surrounding devices, change more often during one multimedia session.

In this context, one of the main motivations for users to use different devices during a multimedia session is precisely the difference in characteristics among the devices, e.g. the larger display of a home theater PC (HTPC), or the mobility of a smartphone despite its smaller display or less available bandwidth. However, typical applications continue to be bound to the device where the application starts execution, and just a few applications implement multimedia session management services.

Ubiquitous computing is the paradigm where computing is made to appear anywhere and anytime, meaning that users, applications and devices are nomadic, and spontaneous interactions are a norm. This paradigm enhances computer use by making many computers available throughout the physical environment while making them effectively invisible to the user. Ubiquitous computing has been discussed since the beginning of the nineties [196], but its vision can be traced back to the mid-1970s [166]. However, we have not yet managed to fully realize it.

Pervasive computing takes ubiquitous computing as a prerequisite and emphasizes (1) mobile data access [41, 143], (2) the mechanisms needed for supporting a community of nomadic users [28, 110] including context-awareness [143], and (3) seamless integration across heterogeneous platforms. Many efforts within the pervasive computing community, distributed systems, and mobile computing have been done to create frameworks [176, 170, 125, 123, 120, 108, 78, 76, 64, 18, 11, 203, 86, 110], platforms [163, 188], middleware solutions [92, 63, 79, 28, 162, 148], programming languages [25, 109], services [149, 43], and protocols [127, 46, 189].

Despite these efforts, the development of applications adhering to the ubiquitous computing paradigm continues to be hard. This situation will continue as long as developers continue to view mobile devices as *mini-desktop* computers, applications as *programs* that run on these mini-desktops, and the application context as a static *virtual space* where a user enters to perform a task and stays there until the task is finished [17]. Therefore, we research how to provide mechanisms to ease the development of mobile multimedia applications that can be separated or joined during a multimedia session. In this way, the applications can use devices with different Input/Output (I/O) communication interfaces to produce or consume multimedia content in the modality (e.g. audio, video, text, or tactile) that is supported by the device, and preferred by the user in a given *context*. This context includes the user physical environment, user preferences, application running environment, and network conditions.

Mobility in ubiquitous computing can be both: *physical*, related to users or devices; or *logical*, related to processes or data. We argue that physical and logical mobility has to be supported by applications. For this, we suggest that multimedia applications should provide services for (1) *user mobility* to detach users from devices, (2) *fine-grained application mobility* to detach applications from devices in a fine-grained manner, and (3) *host mobility* to detach devices from the network access point used to establish a connection.

Solutions for user mobility can be traced back to the late 1950s when John McCarthy described timesharing systems, and user mobility was supported by thin stateless client terminals [185]. This approach, however, requires the application to be pre-installed in a much broader variety of devices today. Moreover, if the application (or parts of it) moves to heterogeneous devices, or if the mobile device changes its point of attachment between networks, the application will need to adapt in many ways.

To the best of our knowledge, modern multimedia applications and Web services provide partial solutions to achieve user, application, and host mobility seamlessly. For example, popular multimedia applications, such as YouTube or Spotify, rely on the ubiquity of Web browsers to implement server-based mobility services at the session layer. However, the use of mainstream browsers has two significant limitations. First, browsers prevent applications from taking advantage of devices without displays. Second, browsers implement a device abstraction layer that prevents the application from processing or using non-standard modalities or peripherals, e.g., haptic devices.

Application mobility relying only on Application Program Interfaces (APIs) from Web services additional limitations. Application developers still must provide an application for each device that the users might want to use, and the users must configure the applications before receiving a redirected multimedia session. At the same time, if the application does not have access to services at the session layer for redirection of multimedia sessions, the mobility of physical or logical endpoint connections of Internet has to be done at a lower layer.

Furthermore, the solutions from popular applications are designed to consume multimedia content, but not to produce and consume content at the same time, such as in video conferencing applications. Thus, we claim that the current mobility mechanisms are not enough for modern multimedia applications to adhere to ubiquitous computing.

This PhD thesis presents the work to provide mobility services at different levels for the development of multimedia applications that adhere to the ubiquitous computing paradigm. The motivation of this work is based on the observation that users of popular multimedia applications are exposed to a dynamically changing set of devices with different form factors and purposes [53]. However, even if the users have the rights to use the surrounding devices, the users cannot take advantage of the devices and their heterogeneity, because most applications today cannot be moved between devices under ongoing multimedia sessions in a seamless manner.

1.2 Problem Statement

Digital multimedia continues to rise in popularity due to (1) the broader availability of multimedia-capable devices, (2) the increase of mobile multimedia-capable devices, and (3) the more access points to the Internet at steadily higher speed rates. However, the development of applications in this paradigm continues to be a hard problem, because the context in which the application will be used is unknown at design time, and it can change at run time. Consequently, users cannot take advantage of the changing availability and heterogeneity of the surrounding devices in a seamless manner.

This thesis addresses four research questions. The reasoning for each question is based on literature research and incomplete evidence from observations of state-of-the-art solutions. In Questions 1 and 3, we refer to the *efficient* property as location-independence, transparency, and seamlessness.

Research Question 1: How can distributed and fine-grained mobile multimedia applications efficiently adapt the production and consumption of multimedia content in the presence of heterogeneous devices and changing user preferences?

If multimedia applications want to produce multimedia content, the internal mechanisms of the application to capture content must be adapted according to the host device and user preferences. Similarly, the presentation of content should be in a suitable and legible format for the hardware that reproduces it.

Research Question 2: How can developers of mobile multimedia applications detach function calls from specific auxiliary software that device drivers or software components provide?

In order for applications to run in heterogeneous devices, the developer should not make any assumptions about the display size or device capabilities, or even that there is a display at all. For example, a video conferencing application may move to the audio system of a car, even if the car does not process the video stream.

Research Question 3: How can mobile multimedia applications continue communication over Internet in an efficient manner, without managing the connection handover at the session layer?

Web services can help to manage connection handover at the session layer, but this approach introduces additional round trips that reduces the time budget for interruptions in multimedia applications. At the transport layer, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) endpoint connections are tightly coupled with the device and network identity, i.e., quadruple of Internet Protocol (IP) address and port at each connection endpoint. This quadruple is broken when the application wants to continue a connection in a different host.

Research question 4: How can distributed components of mobile multimedia applications have access to multimedia content over the Internet, without saturating the host and local network resources, and meeting the strict multimedia deadlines?

Multimedia applications commonly access data in local memory, or from the network at fast enough speed to meet strict multimedia deadlines. In ubiquitous computing, users change devices while using an application, and the needed data by the applications should be readily available in a location-independent manner. If many remote components require the same data from one device, two conditions must be satisfied. First, all components must have a network route that has enough bandwidth and provides high enough end-to-end throughput. Second, the source device must have enough (bandwidth and CPU) resources to serve the data to all other components.

To address these research questions, we propose a middleware and a framework with an API. The scope of the proposals are presented in Section 1.3. Other identified research questions including topics of security, privacy, trust, integration, failure detection, and spontaneous interoperability are described as future work in Section 5.3.

1.3 Scope

This thesis aims to ease the development of multimedia applications that adhere to the ubiquitous computing paradigm. To this end, we focus on how to provide middleware services to application developers, so they are freed from the burden of low-level complexity unrelated to the business logic of their applications. The main services, which we work on in this thesis, provide (1) *adaptation of the processing and collection of multimedia content, i.e., multimedia presentations*,

(2) *mobility of Internet endpoint connections*, and (3) *efficient data distribution for distributed multimedia applications*.

We target distributed mobile applications that produce and consume multimedia content in commodity hardware. By *mobile applications* we refer to fine-grained (i.e., not monolithic) applications that adhere to the application mobility paradigm [204]. We focus on the mobility scheme where users move the application (or parts of it) from one device (source device) to another (destination device) with a *push* policy.

To make possible the delivery of this thesis with the allocated resources, we restrict the amount of work by focusing on scenarios for mobile video conferencing. As an example, think on a user that moves a video conferencing application from the desktop computer at home to the car computer while in transit to work, and again to the meeting room's video conferencing system when arriving at work. Depending on the available devices and user preferences, the user commands the application (or parts of it) to move to the device with the preferred I/O communication interfaces. If needed, the application reconfigures and adapts itself autonomously, allowing the multimedia session to continue seamlessly.

Table 1.1 shows the space of networked applications (a slight adaptation the classification given in [25]). Space I represents the non-distributed, static multimedia applications such as a traditional local chess video game. Space II represents the distributed, static applications such as typical client-server applications or peer-to-peer applications, e.g. Skype. Space III represents the non-distributed, mobile applications designed for load distribution, exploitation of resource (temporal and spatial) locality, or resource sharing, such as monolithic applications using virtual machines or microkernels as in Amoeba and Sprite [59], or VAMNET [32]. Space IV represents the distributed, mobile applications such as YouTube (where mobility is provided by server-based multimedia session management), or applications implemented in Emerald [109] or Obliq [26, 40]. This thesis targets applications in space IV. The detailed requirement analysis for our targeted applications is presented in Section 2.3.

Table 1.1: The space of networked applications

Distributed	**II** (e.g. Skype)	**IV** (e.g. YouTube)
Non-Distributed	**I** (e.g. local chess video game)	**III** (e.g. Migratory microkernels)
	Static	Mobile

1.4 Research Methods

Research methods in computer science are the logical schemes to systematically find well-founded answers to the fundamental question underlying all computing; “What can be (efficiently) automated?” [54]. New findings (as the outcome of applying these methods) are then presented in many forms including theories, algorithms, models, and frameworks for system implementation. In computing, three research methods are defined, i.e., **theory**, **abstraction** (modeling), and **design**; but they are intrinsically intertwined [54].

The research in this thesis has a stronger focus on the design research method. To this end, we follow four research phases (defined in [80]): informational, propositional, analytical, and evaluative. We iterate these phases as we find unexpected results or flaws during the development of the thesis.

The naming and categories of research methods in [54] and [80] differ, but we map the commonalities between the definitions in the following manner. The scientific and analytical methods map to the *theory* method, the engineering method maps to the *design* method, and the empirical method maps to the *abstraction* (modeling) method. The following subsections describe each phase in the research for this thesis.

1.4.1 Informational Phase

In this phase, we do observations on how common multimedia applications are developed and used today, how the mobility of users change their surrounding multimedia-capable devices during one multimedia session, and what are the limitations to take advantage of those devices in a seamless manner. Then, we research state-of-the-art solutions in different computing disciplines. The result of this phase is the problem statement and the research questions (in Section 1.2), goals (in Section 1.3), and requirements (in Chapter 2).

In Chapter 3 we discuss approaches in the research literature that are relevant beyond the Informational phase. After gathering and aggregating the information, and refining the search scope, we stayed updated in different manners. We subscribed to updates from different sources, mainly ACM Digital Library, IEEE Xplore Digital Library, and Google Scholar. We also created query alerts in the Stack Overflow network to receive updates on challenges that application developers of multimedia applications face.

In addition to literature research, the author of this thesis participated in five international academic conferences, and three international industry conferences. The specific participation in the academic conferences consisted in the presentation and discussion of papers [95, 96, 194, 191, 193, 192], and the discussion of this PhD thesis proposal in [139, 186]. The specific partition in the industry conferences was the presentation and discussion of two lightning talks [199, 190], and attendance to [68], where the *Open Media devroom* track is particularly relevant. We also conducted research field by participating to six international hack-festivals [85, 91, 88, 84, 89, 90] related to the development of the state-of-the-art multimedia framework GStreamer. In the next phase, we propose solutions for later analysis and evaluation.

1.4.2 Propositional and Analytical Phases

In this phase, we propose the middleware and the framework illustrated in Figure 4.1 to answer the research questions in Section 1.2. We subdivide the research questions in concerns of the system, and provide an analysis for each concern in Chapter 3. Then, we analyze the proposed middleware and framework in Chapter 4; this analysis is based on data obtained from system performance evaluation of implemented prototypes, and mathematical analysis especially for the proposed parts not implemented.

1.4.3 Evaluative Phase

To evaluate our proposals, we write programs to do performance analysis measurements, and do the mathematical analysis to distinguish from casual observations and validate the non-implemented parts. When results were not as expected, and depending on where any discrepancies arose, we went back to earlier phases (Informational, Propositional or Analytical). Admittedly, the iterations to previous phases could be more exhaustive; however, the evaluation design in this thesis aims at satisfying the answers to the research questions, rather than optimizing the answers.

The nature of each proposed solution demands a different selection of metrics and evaluation techniques. The collection of papers in this thesis evaluates the propositions in the corresponding metrics.

1.5 Main Contributions

The main contribution of this thesis is the realization of the propositions to achieve fine-grained multimedia mobile applications adhering to the ubiquitous computing paradigm, and the knowledge obtained from this realization. The requirement analysis, design, prototype implementation, and evaluation of the propositions that attend to the four research questions identified in this thesis are building blocks in the contributions of this thesis.

The main realized services in this thesis provide adaptation of multimedia presentations by reconfiguring the sequentially connected components that process multimedia streams, i.e., GStreamer *pipelines*, and their topology [193, 192, 191]; migration of TCP and UDP sockets [194]; and data distribution for distributed multimedia components [95]. These services remove the burden of application developers to create multimedia applications adhering to the ubiquitous computing paradigm, thus achieving the aim of this thesis (stated in Section 1.3).

To enable mobile applications to adapt, we conclude that they must be able to self-configure at load and design time. Self-configuration at load time is necessary because application developers should not hardcode their design to specific software or hardware dependencies. Self-configuration at run time is necessary because users' preferences or physical environment can change during a multimedia session, or because the context at the other end of the communication channel can change.

To enable applications to adapt to context changes, we propose to characterize the parameters that represent users' physical environment, user preferences, applications' runtime environment and networks conditions. This approach helps to convey the idea that the construction of multimedia pipelines must take into consideration the current capabilities of the human user as ultimate source or sink component.

Autonomous adaptation of multimedia pipelines is a complex and hard task. It is complex because it requires many low-level mechanisms for memory management, such as clock synchronization, data flow control, component instantiation, and components' state management. It is hard due to the combinatorial explosion when testing the available components in a device. We conclude that architectural constraints mitigate the NP-hardness of combinatorics within the time limits of soft real-time requirements. These constraints are namely the grouping of components by functionality, i.e., functional stages, and the abstraction of sequentially connected components that process a stream to do a certain task, i.e., functional paths, e.g. capture video from a webcam and send it over the network.

The ubiquitous computing paradigm implies context changes including the variation of available or appropriate I/O communication interfaces of devices and users (e.g. hearing or sight impairments due to physical or contextual constraints). To efficiently adapt to these variations, we argue that different adaptation techniques, including *fidelity*, *modality*, *content-based*, and *retargeting* adaptation, are required. Once the combinatorial explosion involved in the autonomous reconfiguration of multimedia pipelines is controlled, applications can take advantage of the adaptation techniques unknown at design time.

The separation of *what* functionality is needed versus *how* this functionality is implemented (just as in the principles of networking protocols and layers or declarative programming), enforces the design of applications that are not hardcoded to specific device software or hardware. As a result, these applications have the *write once, run everywhere* characteristic.

The proposed middleware and API is a step forward making applications mobile, which in turn gives the applications the *install once, configure once* characteristic, and reduces the overhead of users' Personal Information Management (PIM). For applications to provide continuous multimedia services over the Internet, we propose location-independent inter-process communication (IPC) mechanisms between processes running in different devices. For this, we design and implement mechanisms for socket migration as part of the functionality provided by an API that resembles the Portable Operating System Interface (POSIX).1-2008 [182]. Based on results from the evaluation, we conclude that the resources of today's smartphones (i.e., commodity hardware) are sufficient to use these devices as proxies that hide the mobility of endpoint connections by forwarding packets.

In the ubiquitous computing paradigm, users change devices while using the same application. For this to work, the needed data by the applications should be readily available in a location-independent manner. Multimedia applications commonly access data in local memory, or from the network at fast enough speed to meet strict multimedia deadlines. The proposed API provides location-independent data access is an efficient IPC mechanism to share data between processes. If the processes run in different devices, they access the data via an automatically established mesh network and a publish-subscribe (PUB/SUB) [145] service. Based on results from evaluation, we conclude that sharing the multimedia data that devices are already consuming is an efficient approach to aggregate the available CPU and bandwidth resources in ubiquitous computing.

The detailed aspects of these contributions have been peer-reviewed and published in four papers in conference proceedings [192, 193, 191, 194], one workshop paper [95], and one abstract paper [96]. Chapter 4 presents a summary of the papers. Part II includes these papers in the typesetting format of this thesis. For the original publication format, we refer the reader to the publisher's website, specified in the respective bibliography's entry. Additionally, five co-supervised master theses, based on the research questions identified during the research of this thesis, are described in Chapter 4.

1.6 Thesis Structure

There are three parts in this thesis. Part I establishes the context of the thesis, and develops the thread that links the research publications. After this introductory Chapter 1, we describe use cases, assumptions, and the requirements analysis in Chapter 2. Chapter 3 presents the background and related work. The sections in the chapter are titled after the separation of concerns of the research questions. Chapter 4 is the summary of six research papers published in international peer-reviewed proceedings. The author's contributions per paper are stated in the corresponding section. Chapter 5 presents the conclusions, summarizes the contributions, and gives a critical review of the research papers. This chapter also states open issues, future work, and future research.

Part II is the compilation of the research publications that addresses the problem statement. Namely, how to ease the development and use of mobile multimedia applications that adhere to the ubiquitous computing paradigm.

As supportive material, the Glossary, Acronyms, and Symbols sections are prior to Part I. Part III contains an errata and additional use cases.

Chapter 2

Use Cases, Assumptions and Requirements

This chapter presents one use case that helps to exemplify the aim (stated in Section 1.3). Section 2.1 presents one use case and how application developers (using our proposed framework) can implement an application for this use case; this section also details the scope of the targeted application domain. Section 2.2 states the services we take for granted as part of the middleware we propose; these assumptions are out-of-scope work in this thesis. Section 2.3 details the functional and non-functional requirements of the middleware. Section 2.4 summarizes the chapter.

2.1 Use Case – Video Conferencing in Transit

Alice exchanges urgent messages with her colleagues. In many cases, she needs to participate in video conferencing sessions regardless of whether she is at work, at home or in transit. Typically, she is surrounded by different multimedia-capable devices *everywhere* at *any time*. For example, besides having her smartphone next to her, she is surrounded by (1) a desktop computer and a laptop with typical multimedia capabilities and peripherals if she is at home, (2) a display monitor and a car audio system if she is in transit by car, (3) a laptop and a shared CPU in a high performing computer if she is in transit by train, and (4) several dedicated multimedia devices (large displays, high resolution cameras, high sensitive microphones, speakers, joysticks, and haptic devices) if she is at work.

Depending on the available devices and activity, she commands the video conferencing application (or parts of it) to move to the device with the preferred Input/Output (I/O) communication interfaces. If needed, the application adapts and re-configures itself autonomously, allowing the multimedia session to continue seamlessly.

The previous use case (and those in Appendix B) share the following characteristics. Users have access to many different devices with multimedia computing capabilities, including different multimedia I/O interfaces. During one multimedia session, the context (any information that characterizes the user surroundings, preferences, application running environment, or network conditions, which impacts the functional and non-functional requirements of an application) can change. For example, the set of available devices, the device that is preferred or needed, or modalities (audio, video, text, or tactile) for interaction.

In these scenarios, it is impossible for application developers to predict the characteristics of all devices that will surround the user. Instead, we propose an API that developers can use to implement the services by describing them at a high-level. The scope of services we refer to is similar to the following examples. *Render video from the network*, *capture image from a camera and render it in a display*, *capture image from a camera and send it over the network*, *capture audio and render it on speakers*, *capture audio and send it over the network*, and *capture movements from a haptic device and send it over the network.*

The scope of multimedia modalities envisioned in the use cases is *text*, *images*, *audio*, *video*, and *tactile*. The scope of heterogeneous device types includes laptops; desktop computers, and the typical I/O peripherals; dedicated computers as servers; home theater PC (HTPC); smartphones; multimedia devices in cars, trains, airplanes, and refrigerators; joysticks; haptic devices such as the ones in the Da Vinci Surgical System [57]; digital cameras; portable music players; Personal Digital Assistants (PDAs); tables; voice recorders; and GPS devices. Appendix B describes more ambitious use cases, which can leverage on the research from this dissertation, and are expected to become reality in the next decade.

2.2 Assumptions and Out of Scope

The behavior, functionality, and services described in this section are assumed. The design and implementation of the services are out of scope of this thesis.

We assume that the host devices involved in application mobility are connected in an overlay network, as implemented by the coordinator proposed in [95] (summarized in Section 4.6, and included in Chapter 10). Services to open connections through firewalls and Network Address Translators (NAT) are assumed to be implemented.

To achieve application mobility, the computer program should be able to execute in the (possibly heterogeneous) receiving device. In this thesis, we have chosen to develop in the C programming language because most targeted devices usually include C compilers, and manufacturers of I/O peripherals usually provide libraries to be used by programs written in C. Therefore, when we say that an application moves from one device to another, we assume that the source code of the application is moved together with its dynamic state, and a Just In Time (JIT) C compiler in the middleware cross-compiles the code. Then, the middleware imports the dynamic state of the application and resumes execution.

The host devices have the middleware pre-installed; this middleware provides services for code mobility and JIT compilation. The middleware also implements services to monitor the user's physical environment, application runtime environment, and network conditions, i.e., *context*. Privacy and security issues are not taken into consideration in this thesis. The authors in [60, 140] discuss security issues for process migration-aware systems, which is related to the approach in this thesis.

We assume programmers want to concentrate on the business logic of the application to provide usability, high Quality of Service (QoS), and high Quality of Experience (QoE), rather than implementing low-level autonomic mechanisms. We do not design or implement mechanisms for QoS, nor evaluate performance properties of QoE. QoS for multimedia processing is taken cared by internal mechanisms of third-party multimedia components, i.e., GStreamer elements. QoS for communication is left to protocol implementations at the transport or session layer, e.g., TCP, SIP, SCTP or RTSP.

2.3 Requirements

This section defines the expected services, i.e., *service statement*, of the framework to support application mobility, and constraints that the middleware must obey, i.e., *constraint statements*. The service statements constitute the middleware's functional requirements. Functional requirements describe the scope of the services provided by the middleware or Application Program Interface (API). The constraint statements constitute the middleware's non-functional requirements [132].

2.3.1 Functional requirements

We translate the assumed goals of applications developers, i.e., usability and high QoE, as a *safety predicate* based on two requirements. First, the collection of multimedia streams have to be processed on time and in synchrony to a reference clock (skew no longer than tens of milliseconds [177]). Second, the configuration of components has to provide a high enough utility to the user, where utility functions define the user utility. Developers of components of multimedia pipelines define the utility functions.

To satisfy the safety predicate in application mobility, we identify four self-* properties as requirements: **self-adaptive**, **self-configuration**, **self-optimization**, and **self-awareness**. Further requirements are obtained from iterations in the Informational phase (in Section 1.4.1), in particular, from literature [48, 143, 153, 132], and participation in six international hack-festivals of the GStreamer multimedia framework [85, 91, 88, 84, 89, 90]. Next, we state each requirement and give arguments for them.

Self-adaptive: Applications should react to changes in the context by changing their safety predicate accordingly. *Calm computing*, also known as *calm technology* or *disappearing computing*, is the part of ubiquitous computing that talks about removing the distraction of using multi-device applications [198, 197]. Mobile applications should select the appropriate interface, based on the user physical environment, user preferences, application runtime environment, and network conditions. The multimedia presentation selected in this manner is specific to an interface modality and form factor. Particularly distinctive in this thesis, *appropriate selection* does not imply to achieve the highest QoS or use the newest device or the latest network technology[1]. Similarly, think about the users that prefer to watch a movie in their smartphone on the go, instead of staying at the leaving room to watch the movie with higher quality in an HTPC system.

Self-configuration: Applications should react to context changes, and change the connections or components of the application, to restore or improve the safety predicate. Application developers should know *what* services their applications need, without necessarily knowing *how* low-level mechanisms work. In a similar

[1] This assumption attends to the observation from authors in [104] who state that users preferred to turn off the 4G capabilities or their mobile phones when 4G was rolled out, because 3G was more stable, fast enough, and used less battery.

manner, we assume that users are interested in using *what* the services provide, not finding out *how* to configure the application.

Self-optimization: Applications should maximize the utility provided to the user by either maximizing or minimizing the value of a predefined objective function of components. If the user changes her or his environment or preferences, the middleware should treat such changes as a threat to the safety predicate and addresses them. For example, when a DASH (Dynamic Adaptive Streaming over HTTP) component, that proactively checks the available resources, optimizes its parameterization to process the highest bitrate for the given available resources.

Self-awareness: In order to make decisions on adaptation, the middleware should use its context (users' physical environment, users' preferences, users' states, and application running environment) extensively. "A pervasive computing system that strives to be minimally intrusive has to be context-aware, and must modify its behavior based on this information" [166]. It is impossible to predict at any given point in time all future variations in context. Therefore, the collection of context-aware data should be in an open and extensible manner. The framework should provide APIs with abstract services that hide specific devices or sensors.

Persistent communication over the Internet: Communication should resume transparently when an application is moved between devices or networks. The mobility of connection endpoints or point of attachment to a network should not break a multimedia session.

Decentralized solutions: Application mobility should not depend on server-based mobility services. Centralized solutions add time overhead due to round-trip-time, and introduces risks for bottlenecks and one point of failure.

High-level service specification: Application developers should be able to specify multimedia processing as high-level services. It is unrealistic to expect that developers will know how to configure each component needed for multimedia processing, especially when the developers do not know on which devices the applications will be used. Similarly, developers should not invest time on (re-)designing or (re-)implementing mechanisms to enable application mobility.

High-level user preferences specification: Users must inform the application in one way or another what are their preferences or intention, so the application has information to act on, and adapt accordingly, e.g. a user that prefers video over audio or vice versa. Users should be able to provide this information at different levels, because the more technical input from the user, the more expertise from them is needed, and this imposes an entrance barrier for non-expert users. Since user's attention is a limited resource [204], users should not be required to specify a preference for every single parameter or component. That is, the application should be able to adapt with incomplete information from the user.

2.3.2 Non-functional requirements

Non-functional requirements are constraints on the development and implementation of the services we propose to aid multimedia applications to become mobile. The level of adherence to these constraints determines the software's quality. The construction of prototypes helped us to see additional requirements. In this work, we device non-functional requirements as follow.

Freeze time: Program load freeze time should be in the order of hundreds of milliseconds. A tolerable delay in multimedia applications is the amount of time users are willing to wait before giving up on communication. Studies on QoE [16, 21, 160, 105, 97] discuss the service interruption in the order of hundreds of milliseconds as reasonable before users get annoyed. In peer-to-peer applications, where one peer might not be aware of the application mobility action of the other peer, hundreds of milliseconds continue to be valid. However, if the user is aware that an application is being moved, we assume users can tolerate a higher interruption time.

Throughput: The data transfer rate and processing should be sufficient to achieve seamless multimedia processing. Throughput is not only dependent on the available network bandwidth, but also on the packet processing capabilities of the devices involved. Since we can not change the hardware capabilities of devices, our software design and implementation must be efficient to fulfill the soft real-time requirements of multimedia applications. We aim for throughput of at least 1.5 Mbps, because it is the recommended bandwidth for video calling in high definition (HD) in Skype [174], and the recommended broadband connection in Netflix [141], two prevalent popular applications.

General non-functional requirements: We do not discuss explicitly other typical requirements for software development, but we consider them during the design and evaluation of the prototypes. These requirements are *code reusability*, *reliability*, *modularity*, *separation of concerns*, *resource efficiency*, *ease of building*, *ease of deployment*, and *platform independence*.

2.4 Summary

In this chapter, we have described a use case as a hypothetical scenario that can take place at the office, school, home, or while in transit: walking, driving, or being a passenger in public transport. The chapter also outlines functional and non-functional requirements for the proposed services in this thesis.

More services than the ones proposed in this thesis are needed to fully realize the vision of ubiquitous computing. Sections 5.2 to 5.4 describes the identified services that can be built on top of the prototypes and results of this thesis.

Chapter 3

Background and Related Work

In this chapter, we examine the most recent approaches, techniques and thinking in the computing disciplines relevant to fine-grained mobile multimedia applications adhering to the ubiquitous computing paradigm. We summarize the identified gaps in the current solutions, and areas that have remained unsolved. This chapter is the outcome after visiting several times the Informational phase Section 1.4.1, and is the ground for the research questions presented in Section 1.2. The content in this chapter is an additional contribution to the related work of each published research paper.

One could argue that most parts of the systems needed to realize the ubiquitous paradigm have been already developed. However, this paradigm will remain unrealized as long as developers continue to view mobile computing devices as *mini-desktop* computers, applications as *programs* that start and end execution on those mini-desktops, and the application runtime environment as "a *virtual space* that a user enters to perform a task and leaves when the task is finished" [17]. Two projects with the vision to change this view are PIMA (mentioned in [17]), and Mobile Gaia [171], however, these projects are inactive at the time of doing the research in this thesis.

International Business Machines (IBM) was a precursor in autonomic computing [100]. IBM is the company behind the initial proposal of the MAPE-K autonomic adaptation loop model, and developed an autonomic computing toolkit to ease the development of autonomous applications. However, IBM stopped the development of the toolkit in 2004, the source code is unavailable, and based on the information in [102, 103], its latest version did not include mechanisms for autonomic adaptation of multimedia processing.

The motivation of this thesis is similar to the work in the ABLE research group

[51], in particular, the project Rainbow [73, 44]. Architecture-based adaptive systems, such as Rainbow [44], provide features to enforce certain architectural constraints. However, the multimedia adaptation in Rainbow defines the adaptation of multimedia presentations at design time, and adaptation is coordinated at the session layer. Thus, Rainbow's solution does not allow the use of multimedia adaptation types unknown at design time. The authors in Rainbow do not address mobility without changing the standard Internet protocols, nor take into consideration the context of users, applications, and network.

We have proposed the reconfiguration of multimedia pipelines to achieve adaptation of multimedia presentations. Consequently, the type of architectural constraints proposed in Rainbow is impractical for the abstraction of multimedia pipelines as multigraphs.

Aura [176], an architectural framework for user mobility in ubiquitous computing environments, addresses ubiquitous computing by making user tasks first-class entities. Aura claims that resource adaptation is best addressed at task-level; therefore it represents user tasks as a collection of services and context observations that allows tasks to be configured and adapted to the environment. However, Aura does not discuss strict deadlines, which is are a fundamental requirement in our use cases. Examples of tasks as abstract services in Aura are *edit text* and *play video*. In Aura, applications are pre-installed in every device that users can use; this approach, however, does not scale in ubiquitous computing because the applications must be ported and installed on every potential task-receiving device.

Gaia [171] is an operating system that provides a collection of services to manage heterogeneous devices and services. Gaia is designed to ease the development of user-centric, resource-aware, multi-device, and context-sensitive mobile applications. However, Gaia-applications run in the Gaia operating system only; this does not meet our platform independence requirement, and reduces the available libraries and mechanisms that ease the development of multimedia applications.

In summary, we have not found any system that can provide all the mechanisms for multimedia applications to adhere to the ubiquitous computing paradigm. Thus, we proceed to discuss the related work in relation to the concerns of the research questions addressed in this thesis. The concerns are: variability of multimedia presentations, context-awareness, detachment of applications from host devices, adaption of multimedia presentations, decision-making for adaptation of multimedia presentations, mobile inter-process communication (IPC) mechanisms for processes in different devices, data sharing for distributed mobile multimedia applications, reduction of Personal Information Management (PIM) overhead in multi-device applications, and scalability issues in ubiquitous computing.

3.1 Variability of Multimedia Presentations

The autonomic construction of multimedia pipelines introduces a variability problem with combinatorial (NP-hard) complexity. Reducibility in combinatorial problems has been much addressed [118, 152, 36, 12, 150, 151, 112, 35], and in theory, many heuristic techniques can limit the variability problem in the multimedia pipelines abstraction, i.e., directed graphs. However, autonomous linking and reconfiguration of multimedia components are complex tasks in themselves, especially in fine-grained mobility scenarios, where each media stream should be able to be processed in different devices while in synchrony to a clock.

The authors in [61] address three challenges: heterogeneity, variability and efficient delivery of video in content-based networks. Their adaptation scheme creates different multimedia representations using scalable coding in content-based network overlays, and they use multi-dimensional utility-functions for video selection. Utility functions take as arguments QoS: temporal (framerate), luminance and chrominance quality; user preferences: region of interest; and available resources: network bandwidth and CPU utilization. The authors implement a publish-subscribe (PUB/SUB) [145] protocol as an alternative to adaptive bitrate streaming systems to select a different version of the video. Also, their system limits QoS variations across image regions by constraining the quality deviation. However, their proposal does not handle delay or seamless adaptation, and does not discuss when or how a receiver should transit from one configuration to another. Thus, to the best of our knowledge, none of the papers above cited address variability of context-aware systems and complexity of multimedia pipelines to make the decision on which configuration is the most suitable for a given situation.

3.2 Adaption of Multimedia Presentations

We argue that the needed adaptation type for multimedia applications adhering to the ubiquitous computing paradigm is unknown at design time. To understand the complexity of adaptation of multimedia, and the coverage of related work in this topic, we first define the terms *multimedia*, *multimedia content*, and *multimedia presentation*. Then, we illustrate the different types of adaptation in multimedia in

Figure 3.1. Figure 3.2 maps these adaptation types to the multimedia modalities targeted in this thesis.

Multimedia is any *collection* of data including text, graphics, images, video (moving images presented as a sequence of static images), audio, tactile modalities or any system for processing or interacting with such data. Multimedia content is something (e.g. a person, object or scene) selected by, e.g. an artist, a photographer or multimedia developer, for multimedia representation . A multimedia presentation is multimedia content composed by a collection of media.

There are many types of adaptation of multimedia data. Figure 3.1 shows a diagram of how adaptation types are related. For example, an application can select a specific modality, say video, and then apply any of the three fundamental adaptation types: spatial, temporal, or quantization. Spatial adaptation applies to modalities with graphic content, and further adaptation can be applied: complexity, content, and other. Spatial adaptation includes retargeting algorithms, such as the one described in [161] (summarized in Section 4.8.1). Complexity adaptation combines spatial, temporal or quantization adaptation [167]. Content adaptation analyzes the data to determine, for example, detect important regions and remove those less important.

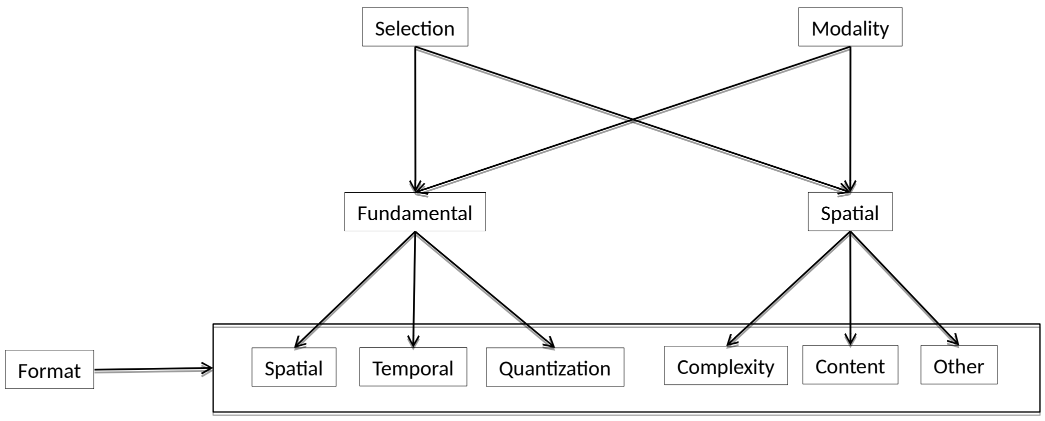


Figure 3.1: Types of adaptation of multimedia data

Figure 3.2 shows the relation between media modalities and the type of adaptation that can be applied to them. For example, temporary adaptation can be applied to video by changing the number of frames per second, or audio by changing the sample rate, but it does not make sense to apply it to text modality. An example of other modalities is tactile produced or consumed by joysticks or other haptic devices.

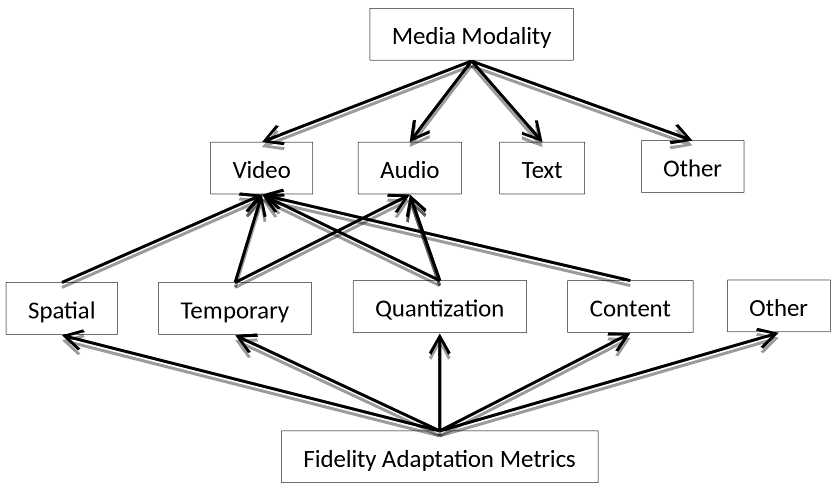


Figure 3.2: Types of media modalities and their adaptation types

Modality selection by merely ignoring already processed media streams, but not stopping its processing, results in a waste of relatively significant amount of resources. This waste of resources is especially relevant when the video modality is processed without anyone consuming it. To evaluate the relevance of this waste of resources, we measured[1] the amount of CPU time and data processed to reproduce video and audio from a file in a video player. The measurements show that the processing components for video consume about 80% of CPU time, and 97% of the processed data (already decompressed).

WebRTC [189] continues to create the expectation that it provides a one-fit-all adaptation mechanism (including mobile applications in the ubiquitous computing paradigm). WebRTC, however, targets multimedia to be consumed on the Web, and the typical content is usually authored for presentation on specific platforms. The main components of WebRTC include several JavaScript APIs for multimedia presentation. JavaScript expands the capabilities of Web browsers, but it cannot take advantage of specific Input/Output (I/O) communication peripherals in heterogeneous systems. This limitation was mentioned in the year 2000 by [17], and continues to be true at the time of writing this thesis. Therefore, JavaScript and the use of Web browsers as a platform for the development of multimedia applications represent a step backward in ubiquitous computing. In summary, related work on adaptation of multimedia presentations covers only a subset of the adaptation types presented in Figure 3.1.

[1]Source code for multimedia pipeline profiler available on `https://gitlab.com/francisv/gst-instruments`

3.3 Decision-making for Adaptation of Multimedia Presentations

GStreamer [86] implements rules to automatically build pipelines (or parts of them) in GStreamer elements such as *decodebin*, *playbin*, and *encodebin*. The components used to build the pipeline are selected according to the rank assigned to each component; this rank is assigned by consensus among the core developers of the GStreamer framework. The input and output connectors of the components (*pads* in GStreamer terminology) are instantiated or created based on the input stream or sink component selected by the application developer at design time. The parameterization of the components and connectors is set according to the input stream or selected sink component, and the order of enumerated values for the parameters. The rules in the GStreamer elements for automatic pipeline building are, however, not enough for autonomous mobile applications for the following three reasons. (1) They build only one pipeline, meaning that all other possible valid pipeline variants are untested, and it is taken for granted that the only built pipeline is the one that provides the highest utility to the user. (2) The selection of components in the pipeline does not take into consideration the user physical environment, user preferences or network conditions. (3) The automatically built pipelines cannot be re-configured.

The Adaptation Management Framework (AMF) proposed by [169] provides a dynamic adaptation mechanism to automatically adapt content and services to a user's current capabilities, i.e., device and environment capabilities and their individual preferences. AMF uses graphs to represent their adaptation configurations, and uses the Dijkstra algorithm to determine the shortest path through the graph from the node representing the original content to the node representing the adapted content. However, the decision-making algorithm of AMF aims at optimal service based on time and service cost (price) only.

Jannach et al. [108] present a decision-making algorithm as planning sequences. They measure effectiveness based on the number of steps of the adaptation plan, not on properties of adaptation types. The reasoning engine works with arbitrary sets of predicate symbols. Consequently, the introduction of new types of predicates does not require changes in their planning algorithm. Their algorithms try to maximize the user's experience for a given environment, i.e., device capabilities, network conditions, and user preferences. They use three methods to call adaptation mechanisms: Web services interfaces, Java interfaces, and dynamic invocation of C/C++ implementations. However, the proposed mechanisms are implemented within the application, and the description of multimedia content and available adaptation mechanisms is accessed from a server. Furthermore, the authors do not analyze the processing time of their solution, and they do not discuss any scalability issue.

3.4 Context-awareness

We argue that the use of contextual information provided by sensors in multimedia-capable devices is adequate to make an autonomous decision on how to adapt. This decision is based on research in the semantic gap [107, 106], ubiquitous computing [196], user-centric systems [171, 52, 37, 121, 52], and context-aware systems [55, 176, 67, 56, 110, 153].

The semantic gap in multimedia processing states that there is a disparity between the stored multimedia content and the information that multimedia systems have to process that content in a meaningful way. The authors in [107, 106] propose that in order to process multimedia content in a *meaningful* manner, the system in question should use context (metadata) information to act accordingly. For example, if the system knows that the data being processed is of a person speaking in a room, and if an available video retargeting component, e.g., SeamCrop, is annotated as a good match for video conferencing in rooms, then the system is able to autonomously select the SeamCrop and perform retargeting adaptation for small displays. In [196], Weiser claims that if computers use context, such as location, they can adapt their behavior in significant ways without requiring even a hint of artificial intelligence. Thus, the extensive use of context and metadata annotation aid to close the semantic gap.

To prove the feasibility of this contextual approach, we analyze the readily available source of context in our targeted application domain. In Unix systems, the *proc* pseudo-filesystem [34, 142] provides an interface to kernel data structures, which contains the context of the process (running environment including information on I/O peripherals; availability, and configuration of hardware, and software resources). Mobile phones include devices to collect information about the user environment, such as GPS, accelerometer, microphones, and light sensors.

Quality of service-aware component Architecture (QuA) [8, 79] uses run-time models, utility functions [2], QoS prediction, and service planning. QuA does not provide a comprehensive context management middleware but may use a context middleware similar to the one provided by MADAM [74, 3] or use the context model of QuAMobile [6, 7]. However, neither QuA nor QuAMobile adapts applications based on different device capabilities at run time, but only bandwidth fluctuations.

Some works have relied on ontologies, and the metadata specification of MPEG-7 and MPEG-21 Digital Item Adaptation (DIA) framework to model context. [76, 93, 92, 205, 146, 43, 130] use ontologies, [205, 20] use MPEG-7, and [130, 20, 201, 123] use MPEG-21 DIA. Although we could reuse some of these models, they yield to much higher complexity than ours. Also, the query time in ontology-based context models is over 100 ms in all reported results. Therefore, we regard the overhead of ontologies as impractical for our use cases. In addition, MPEG-21 DIA considers only a narrow set of user preferences for multimedia selection and adaptation [155]. In summary, we do not base our framework in such standards because they do not address the problem of dynamic and autonomous adaptation.

3.5 Detachment of Applications from Host Devices

Application mobility [204] gives users the freedom to decide where applications should execute, and introduces opportunities for augmenting the available resources including different I/O communication peripherals, memory and communication channels. However, applications are usually only developed and tested for a specific device class, e.g., smartphones, and have to be adapted for other device classes, e.g., home theater PC (HTPC) or laptop. Also, typical auxiliary software from device drivers or software components tightly couples the application to the device, which creates a portability problem.

Application mobility has been addressed in many forms by the distributed computing community; these forms include process migration, remote execution, cloning processes, object migration, code mobility, and mobile agents [140]. Process migration is the technique where an active process is moved from one machine to another. After migration, the process must continue normal execution and communication. The original motivation for process migration was the resource scarcity, reduce the burden of system administration, user time sharing in expensive mainframes, workload distribution, and fault resilience, especially for long processing tasks, where the unexpected or force termination represented an unacceptable or very expensive lost of data. By contrast, the motivation for application mobility in this thesis is to enable users to take advantage of the surrounding heterogeneous multimedia-capable devices during a multimedia session.

Table 3.1 presents a comparison of characteristics of applications in traditional process migration and application mobility for multimedia applications. The differences shown in the table make very difficult to directly apply findings from previous work in process migration onto the development of mobile multimedia applications adhering to the ubiquitous paradigm.

Table 3.1: Differences of characteristics between traditional process migration and application mobility for multimedia applications

Characteristic	**Traditional process migration**	**Application mobility for multimedia applications**
Component	Monolithic tasks	Polylithic tasks
Freezing time	No specific constraints	Under hundredths of milliseconds
Media type	One media type	Multimedia
Interaction	Non-interactive	Interactive
Device	Homogeneous	Heterogeneous

Transparent remote execution in the presence of heterogeneous devices requires support that is as complex as in transparent heterogeneous process migration [175]. Remote execution does not work in case a user wants to move the entire application without leaving dependencies in the source device.

Recent efforts in cloning processes are CloneCloud [45] and DPartner [208], which exploit clouds infrastructures to mitigate resource poverty of mobile devices. However, they do not address multimedia applications requirements and the applications developed in those systems cannot access the differences in I/O capabilities of devices because those differences are hidden away by their virtual machines based on the JVM.

Object migration was initially designed for small-scale and homogeneous local area networks [204]. Emerald [109] and Obliq [25] are programming languages designed with object migration capabilities. However, the low adoption of these programming languages results in no available components related to the adaptation of multimedia presentations.

Code mobility is assumed to operate in large-scale and heterogeneous networks [204]. Various design paradigms, such as code on demand, remote evaluation, and mobile agents, have been proposed to enable code mobility [70]. However, code mobility or mobile agents that are implemented in languages such as

Java (e.g. [81]), Telescript, or Tcl/Tk [140], lose access to device heterogeneity due to hardware abstraction at language-level.

In summary, proposals for application mobility from distributed computing are helpful in providing underlying migration services, but inadequate to detach applications from devices and provide seamless application mobility for multimedia applications in ubiquitous computing. Moreover, to the best of our knowledge, we are unaware of current efforts from the industry on detachment between applications and devices. To the contrary, companies such as Apple continue to develop device-specific applications as a competitive advantage, e.g., Facetime, iTunes, and iMovie, which represents a step backward in ubiquitous computing.

3.6 Mobile IPC Mechanisms for Processes in Different Devices

The mobility of Internet endpoint connections has been widely researched [127, 147, 31, 30, 137, 24, 99, 117, 207, 71] (discussed in Section 9.5). However, to the best of our knowledge, no connection handover system provides the requirements needed for moving Internet endpoint connections of multimedia applications in our use cases. Namely, low handover time, high throughput, legacy application support, portability, and independence from special infrastructure support.

More recent efforts by Apple, the *Network.framework* [82], moves the transport layer from kernel to user-space, just as SOCKMAN (summarized in Section 4.5 and included in Chapter 9) does. Network.framework implements mechanisms for transparent connection reestablishment, but does not support legacy applications. Moreover, their solution is supported only in the operating systems iOS, macOS, and tvOS, i.e., operating systems from Apple; these operating systems are tailored for devices from Apple, and their source code is closed and proprietary.

3.7 Data Sharing for Distributed Mobile Multimedia Applications

Ubiquitous computing implies a variety of hardware with different architectures and its private memory. If components of multimedia applications are expected to move to different devices, the middleware must include mechanisms to share the needed data stream over the network. The main paradigms for data sharing in distributed systems as middleware are document-based (Web pages), file-system-based, object-based, and coordination-based [181]. The first two paradigms are not relevant to our target domain (e.g. video conferencing applications), because the shared data include streams, not only (static) documents or files.

The most relevant related work in object-based middleware approaches for our uses cases is Common Object Request Broker Architecture (CORBA) [195, 172]. CORBA is a specification that provides a standard message interface between distributed objects, and it has been used to implement object-based middleware [202]. It has been defined by experts in the field of mobile components, but it has failed to enable universal interoperability due to its complexity [28].

Related work in coordination-based middleware (presented in Section 10.2) discusses Linda [75], LIME [156], SPREAD [49], Munin [23, 42], and the scalable tuple space model in [47]. These systems propose different mechanisms to achieve Distributed Shared Memory (DSM) [128, 129], but none of them can provide the low latency needed for multimedia applications.

3.8 Reduction of Personal Information Management (PIM) Overhead in Multi-device Applications

People use multiple devices due to different form factors and modes of interacting (keyboard, mouse, stylus, finger, or multi-touch), device portability, task completion time, having one computer for work and another for home, software and operating system differences, or transitioning from an old device to a new device [53]. The overhead of PIM is reduced if the user does not have to install and configure the same application in all devices that the user wants to use. Our motivation for reduction of PIM is similar to PIMA in [17]. However, their project is server-based and uses Java for implementation. Consequently, the required Java Virtual Machine (JVM) prevents application mobility to take advantage of specific I/O peripherals. Moreover, PIMA does not take into multimedia processing, and the project is inactive at the time of doing the research for this thesis.

Another way to reduce the overhead of PIM is the use of servers as in Cloud4all [146] or practically any other cloud services from the industry. Cloud computing approaches can be of aid to the application mobility approach, but cloud computing is not sufficient because the applications must be pre-installed.

Otherwise, it defeats the purpose of seamless use of devices in ubiquitous computing.

3.9 Scalability Issues in Ubiquitous Computing

The inherent variability of availability and differences of devices, and changes of user context and preferences in ubiquitous computing demands applications to adapt in uncountable ways. We address this variability as a variability management problem in Dynamic Software Product Line (DSPL).

The authors in [35] address combinatorial complexities of variability management with heuristics methods. The authors describe a greedy approach to calculate the utility of only promising variants, and discard the evaluation of functions with *low* weight values. In this approach, utility functions are divided into a *stable* and an *unknown* part. The stable part is evaluated, and only those variants with an already known *high* utility value are kept. Low and high values are defined by the application developer, component developer, or user, or can be calculated based on the distribution of known utility values in the search space. As a result, the unknown part is evaluated only for those variants with already known (stable) utility values. Consequently, the number of evaluated variants is reduced into the linear domain.

The authors in [1] propose models to enforce multi-constraint that allows the shortest path to be found in polynomial time. Their approach is validated through a real-world example implementing adaptive scenarios in the domain of mobile computing. For this purpose, the authors propose linear programming, in particular, simplex or interior-point methods.

3.10 Summary

In this chapter, we have discussed the background and related work on the concerns addressed in this thesis. The content in this chapter presented additional related work to what is discussed in Part II. Each of the efforts described in this chapter can be applied on application mobility adhering to the ubiquitous computing paradigm, but they only address a piece of a larger puzzle. We conclude that the difference in motivation, design decisions, and implementation approaches in these efforts, make very challenging to integrate previous solutions.

Chapter 4

Summary of Research Papers and Author's Contributions

This chapter describes the relationship between the problem statement (stated in Section 1.2), the aim (stated in Section 1.3), and how the published research papers (included in Part II) contribute to solve the problem statement. The author's contributions are stated in the corresponding section of each paper in this chapter.

The complexity of integral solutions for ubiquitous computing makes very difficult to draw a line on the area where each research paper contributes to answer the research questions. In order to see the relation between each research question and each paper, we map specific concerns in the questions, the papers, and the MAPE-K phases in Table 4.1. The table also contains five co-supervised master theses, whose projects were created as a result from the ongoing research of this PhD thesis. Mnemonics starting with 'P' refer to research papers, mnemonics starting with 'M' refer to the master theses.

P1 to P6 are summarized in Sections 4.2 to 4.7 and included in Chapters 6 to 11. The master theses M1 to M5 are summarized in the context of this PhD thesis in Sections 4.8.1 to 4.8.5. For the full text of the master theses, we refer the reader to the citation in the corresponding section.

In Table 4.1, we make a distinction on when the solution is to be used: at design, load or run time. We also group the parts that belong to the framework or the middleware.

P1 addresses Research Questions 1 and 2. It addresses the detachment of applications from their host device and scalability issues. It is part of the Plan and Execute phase in the MAPE-K model, and the middleware solution at run time.

Table 4.1: Overview of research work in relation to research questions, separation of concerns, MAPE-K phases, framework and middleware

		Framework		Middleware	
Research question	**Concern**	**MAPE-K**	**Design time**	**Load time**	**Run time**
1	Variability	Plan	P2	P3	
1,2	Context	Monitor, Analyze	P2	P2	
2	Detach application from device	Execute	P2, P3 P4, P5 P6, M2		P1, P2, P3 P4, P5, P6 M2
1	Adapt multimedia presentation	Analyze, Plan, Execute	P2	P3	P1, M1, M3
1	Decision-making	Plan	P3, M2	P3	P3, M2
3	IPC in different device	Execute	P4, P6 M5	P4, P6	P4, P6, M5
4	Data sharing	Execute	P5, P6	P5, P6	P5, P6
1	Reduce PIM hassle	Execute	M2		P6, M2
1	Scalability issues	Plan, Execute	P2	P2, P3	P1

P2 is the most comprehensive paper in this thesis. It addresses Research Questions 1 and 2. It addresses the variability of pipeline configurations, context-awareness, detachment of applications from host devices, adaptation of multimedia presentations, and scalability issues in the autonomous creation of the variability search space. It is part of the all phases in the MAPE-K model, the framework and middleware design at design, load and run time.

P3 addresses Research Questions 1 and 2. It addresses the variability of configurations, the detachment of applications from host devices, the adaptation of multimedia presentations, decision-making, and scalability issues. It is part of all phases of the MAPE-K model except the Monitor phase, the framework and middleware at design, load and run time.

P4 addresses Research Questions 2 and 3. It addresses the detachment of applications from host devices, and IPC mechanisms between distributed multimedia components. It is part of the Execute phase in the MAKE-K model, the framework and middleware at design, load and run time.

P5 addresses Research Questions 2 and 4. It addresses the detachment of applications from the host devices, and data sharing between distributed multimedia components. It is part of the Execute phase in the MAPE-K model, the framework and middleware at design, load and run time.

P6 gives an overview of all research questions. It addresses the detachment of applications from host devices, IPC mechanisms and data sharing between distributed multimedia components, and the reduction of hassle in Personal Information Management (PIM). It is part of the Execute phase in the MAPE-K model, the framework and middleware at design, load and run time.

M1 addresses Research Question 1. It addresses the concern on how to adapt multimedia presentations. It is part of the Analyze, Plan, and Execute phases in the MAPE-K model, and the middleware at run time.

M2 addresses Research Questions 1 and 2. It addresses the concerns of detachment of applications from host devices, decision-making, and reduction of PIM hassle. It is part of the Plan and Execute phases in the MAPE-K model, the framework and middleware at design and run time.

M3 addresses Research Question 1, more specifically how to adapt multimedia presentations. It is part of the Analyze, Plan, and Execute phases in the MAPE-K model, and the middleware at run time.

M4 is a proof of concept of the design decision for component-based applications as mobile applications and the middleware. We discuss it when we describe Figure 4.1.

M5 addresses Research Question 3 on how to enable inter-process communication (IPC) mechanisms in distributed components of multimedia applications. It is part of the Execute phase in the MAPE-K phase, the framework at design time, and the middleware at run time.

The proposed middleware relies on the hardware abstraction provided by the Application Program Interface (API) of device drivers, and offers an API, which encapsulates the mechanisms for the autonomous adaptation of the application as part of a framework. The offered API reduces the burden of developing mobile multimedia applications. The middleware behaves in a resource-aware manner by adapting the processes accordingly. Figure 4.1 presents the relation between the research papers and the master theses in the middleware or application layer.

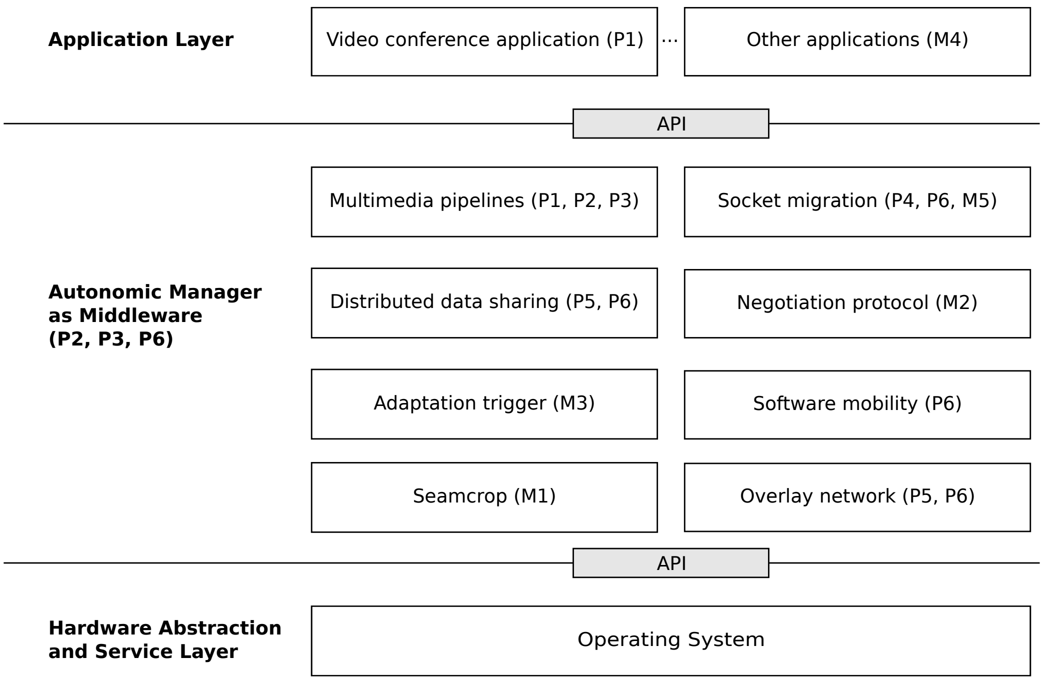

Application Layer
Video conference application (P1)
...
Other applications (M4)
API
Multimedia pipelines (P1, P2, P3)
Socket migration (P4, P6, M5)
Autonomic Manager as Middleware (P2, P3, P6)
Distributed data sharing (P5, P6)
Negotiation protocol (M2)
Adaptation trigger (M3)
Software mobility (P6)
Seamcrop (M1)
Overlay network (P5, P6)
API
Hardware Abstraction and Service Layer
Operating System

Figure 4.1: Overview of research work and proposed architecture as middleware

P2, P3, and P6 discuss the design and implementation of the autonomic manager as middleware. P1, P2, and P3 present implementation of dynamic reconfigurable multimedia pipelines at the middleware layer. P1 also implements a prototype to test the proposed solutions. P4, P6 and M5 present the design and implementation of socket migration service. P5 and P6 describe the design and implementation of two services, the distributed data sharing by multimedia components, and needed overlay network to have access between the involved devices.

P6 also introduces the approach of software mobility, i.e., application mobility, taken in this thesis.

M1 presents the design and implementation of the service to do online video retargeting in the Seamcrop service. M2 presents the design and implementation of the negotiation protocol to move applications between heterogeneous devices. M3 describes the design and implementation of the service that starts the adaptation of the application, adaptation can be triggered by context changes in the host device, or after an application has been moved to another device. Finally, M4 is a proof of concept of a component-based application for application mobility.

All the services illustrated in Figure 4.1 should work properly when used by one application. However, due to time limitations, we have not verified this assumption. This verification is an open issue in the current state of the thesis.

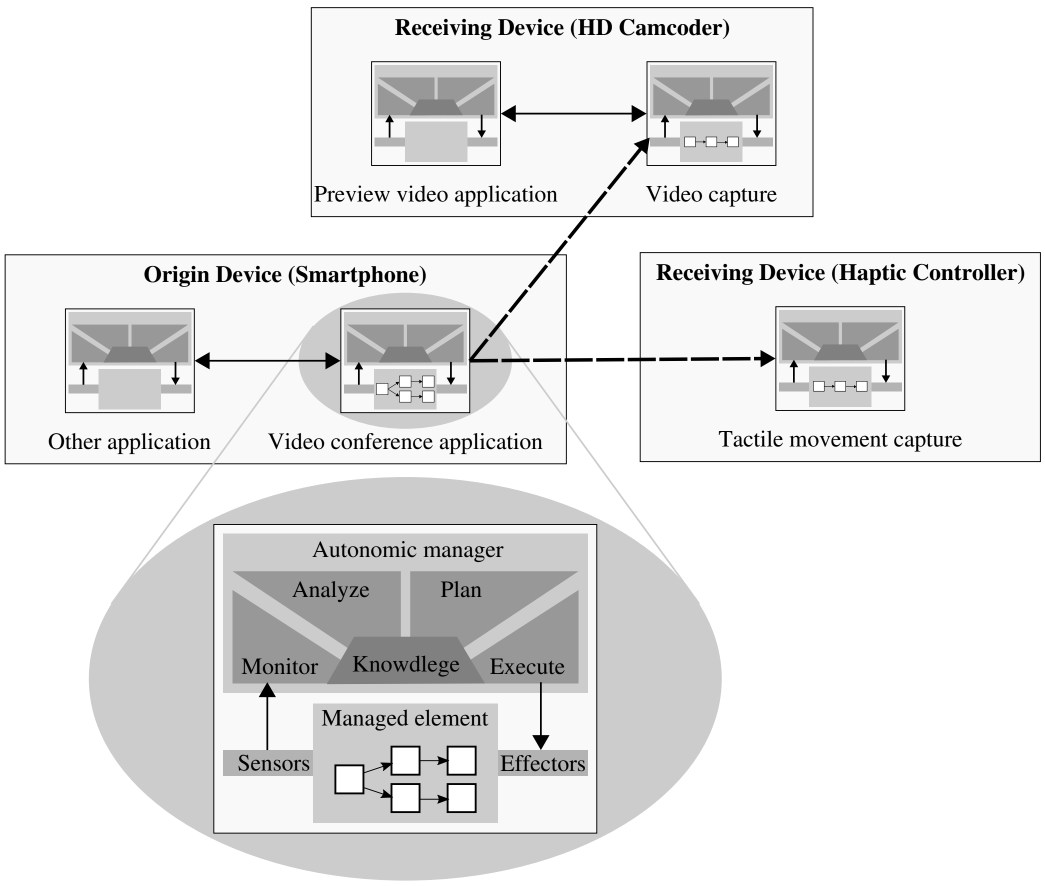
The services here discussed are by no means all the needed services to achieve seamless application mobility. Other identified services include a Just In Time (JIT) C compiler to achieve portability and mobility of programs, and a component to enforce security between connections in the overlay network. Moreover, the library of each service, can be also seen as a mobile component, and it should be able to move to another device just as a mobile application using the middleware. These services and their self mobility, however, are not in scope of the thesis but will be work for the future.

4.1 TRAMP and MAPE-K

The main motivation of this thesis is to enable users to take advantage of the changing availability and heterogeneity of devices in the ubiquitous computing paradigm. For this, we proposed the architecture of TRAMP Real-time Application Mobility Platform (TRAMP) [96] in an early stage of this thesis. However, work on the challenges addressed in this thesis showed the need for a clearer separation between the middleware (initially proposed in [96]), the framework and APIs offered to application developers.

After revisiting the Informational phase (described in Section 1.4.1) several times, we propose an evolution of TRAMP that follows the MAPE-K autonomous control loop model [113]. We called the new proposal as Dynamic Adaptation of Multimedia Presentations in Application Mobility (DAMPAT) [191]. Therefore, papers [192, 193, 191] refer to DAMPAT, whereas papers [194, 95, 96] refer to TRAMP.

Figure 4.2 shows components of mobile multimedia applications as *managed elements*, and how they can move in a fine-grained manner to different devices. The solution contains an *autonomic manager*; a software component configured by application developers using high-level goals. This manager separates the concerns of the challenges in four phases: *Monitor*, *Analyze*, *Plan*, and *Execute*, which create and share information (*Knowledge*). The manager uses the monitored data from *sensors* and created knowledge in the system to analyze, plan and execute the low-level actions that are necessary to achieve the goals specified by application developers and users. The *effectors* apply the actions. A more detailed description of the MAPE-K model is in Section 7.2.1.



Receiving Device (HD Camcoder)
Preview video application
Video capture
Origin Device (Smartphone)
Other application
Video conference application
Receiving Device (Haptic Controller)
Tactile movement capture
Autonomic manager
Analyze
Plan
Monitor
Knowdlege
Execute
Managed element
Sensors
Effectors

Figure 4.2: Proposed solution as distributed autonomous adaptation loop

4.2 P1 – Dynamic Adaptation of Multimedia Presentations for Videoconferencing in Application Mobility

Authors: Francisco Javier Velázquez-García, Pål Halvorsen, Håkon Kvale Stensland, and Frank Eliassen

Authors' Contributions: Velázquez-García did the design, implementation, evaluation, and writing of this paper. Halvorsen, Stensland, and Eliassen supervised the work involved to publish this paper. Supervision included discussion on arguments, paper structure, and text to improve the clarity of some parts of the paper.

Reference in Bibliography: [193]

Included in: Chapter 6

Address or Attends to:

- Research Questions 1 and 2 (stated in Section 1.2).
- Concerns in the Plan and Execute phases on how to detach mobile applications from host devices, how to adapt multimedia presentations, and how to address scalability issues (see Table 4.1).
- Adaptation at run time handled by the middleware (see Table 4.1).
- Prototype[1] as middleware and application layers (see Figure 4.1).

Summary and Thesis Relevance

For multimedia applications to adhere to the ubiquitous computing paradigm, we propose application mobility. We argue that application mobility mitigates the overhead of PIM if the applications are designed and implemented in a way that: (1) applications provide self-adaptive, self-configuration, self-optimization,

[1]Source code available on `https://gitlab.com/francisv/gstreamer-prototypes` and `https://gitlab.com/francisv/gstreamer-devel-tests/blob/ubuntu_branch/tests/benchmarks/complexity.c`

and self-awareness properties, and (2) users have the means to provide their preferences without expert knowledge of the properties of the multimedia pipeline configuration.

The self-adaptive property in application mobility implies adaptation in two aspects. First, the internal configuration of the application should self-configure according to the available software and hardware Input/Output (I/O) interfaces. Second, the presentation of multimedia content has to adapt to the user preferences and the user's physical environment.

In paper [191] (summarized in Section 4.4 and included in Chapter 8) we introduced the proposal to achieve adaptation by configuring multimedia pipelines. For this, we investigated and designed architectural constraints to control the NP-hardness of combinatorial growth caused by compositional and parameterization variability when autonomously testing and linking multimedia pipelines. However, it remained a concern on the overhead time to execute the Plan phase (in the Monitor, Analyze, Plan, and Execute (MAPE) model); this overhead time can introduce intolerable delays if introduced precisely when the adaptation is needed.

In this paper, we investigate how to improve the efficiency of the autonomous adaptation loop by reducing the time spent in the Plan and Execute phase. For that purpose, we measured the time spent in different steps when building multimedia pipelines in the GStreamer multimedia framework. The variable factors in the measurements are the pipeline topology, the number of components in the pipeline, and the number of processed buffers in the pipeline. The instance of measurements in Figure 4.3 shows that component instantiation spends the longest amount of time.

The instantiation time of each component is longer if the component requires services from hardware. Also, the number of queries involved to negotiate the capabilities of components is dependent on the query handlers implemented in each component. Thus, the unpredictability of time spent by queries, as presented in Tables 6.1 and 6.2. Based on these observations, the goal of this paper is to avoid the re-instantiation of components and duplication of queries as much as possible.

The proposed approach presented this paper is to reconfigure multimedia pipelines (change topology or components) in the GStreamer framework while keeping the instantiated components in memory as long as they will continue to be tested. A direct effect is the reduction of time spent in re-instantiating components and handling duplicated queries.

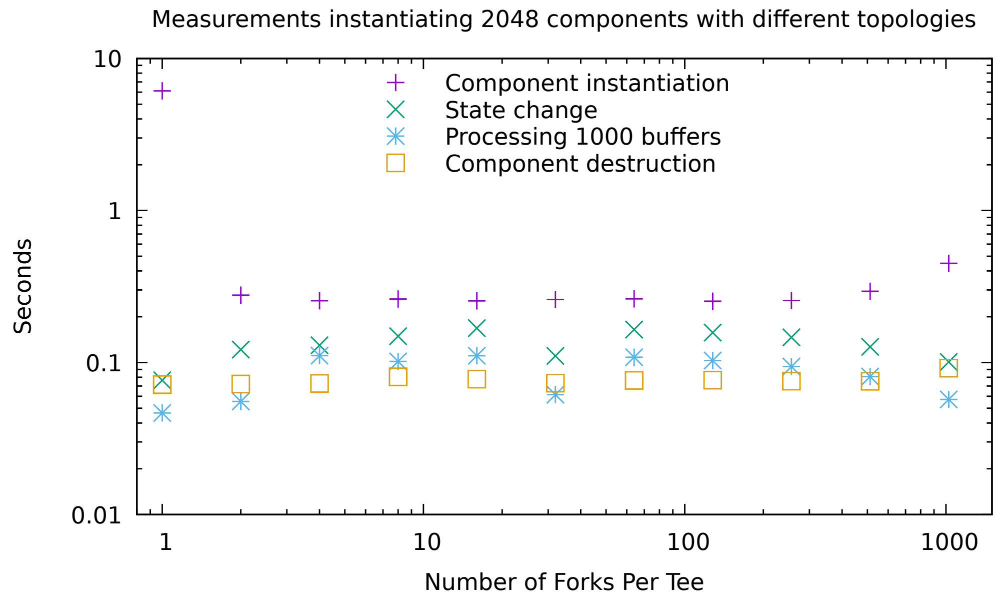


Figure 4.3: Measurements of time spent to build GStreamer multimedia pipelines. The plot shows the time to instantiate and destroy GStreamer components, change their state, and process n number of buffers in the entire pipeline. To change the topology, the number of source connectors (forks) per component increases in quadratic order. The axes are in logarithmic scale.

4.3 P2 – Autonomic Adaptation of Multimedia Content Adhering to Application Mobility

Authors: Francisco Javier Velázquez-García, Pål Halvorsen, Håkon Kvale Stensland, and Frank Eliassen

Authors' Contributions: Velázquez-García designed, implemented, and evaluated the work presented in this paper. Halvorsen, Stensland, and Eliassen co-supervised the work on this paper by debating the arguments in the paper, discussing structure and arguments flow; they also suggested text to convey the content in a clearer manner.

Reference in Bibliography: [192]

Included in: Chapter 7

Address or Attends to:

- Research Questions 1 and 2 (stated in Section 1.2).
- Concerns in Monitor, Analyse, Plan and Execute phases on how to address variability of multimedia pipelines, how to model the context that has impact producing or consuming multimedia content, how to detach mobile applications from host devices, how to adapt multimedia presentations, and how to address scalability issues (see Table 4.1).
- API to be used at design time provided as a framework, and adaptation at load and run time handled by the middleware (see Table 4.1).
- Mathematical analysis of utility functions and prototype[2][3] middleware (see Figure 4.1).

Summary and Thesis Relevance

In [96] (summarized in Section 4.7 and included in Chapter 11), we introduced the initial research questions of this thesis and the initially proposed middleware. Further research during the work of this thesis brought the need for a clearer separation of concerns to address the complexity of autonomous systems. For this purpose, we implement an autonomic adaptation loop following the MAPE model (see Section 4.1). We call the result middleware and framework as DAMPAT initially introduced in [191] (summarized in Section 4.4 and included in Chapter 8). In this paper, we present the holistic motivation, design, implementation, and evaluation of DAMPAT.

We revisit the autonomous (*self-managing*) properties required to achieve application mobility adhering to the ubiquitous computing paradigm. The identified self-managing properties are self-adaptive, self-configuration, self-optimization, and self-awareness.

In [191], we illustrate how developers can specify a group of components by functionality in what we call *functional stages* at three different levels of knowledge (see Table 8.1). In this paper, we define the term *functional path* as an abstraction of the sequentially connected components that process a stream to do a certain task, e.g. capture video from a webcam and send it over the network. Functional paths allow application developers to provide a clearer intention in a

[2]Source code available on `https://gitlab.com/francisv/gstreamer-prototypes`.

[3]Implementation of context models as MySql relational database models available on `https://gitlab.com/francisv/dampat`.

Table 4.2: Enforcement of functional path combinations using the BRGC algorithm. In this example, there are two functional paths involved: W_1 and W_2. The functional path W_1 has two possible pipeline configurations w_1, w_1', and the functional path W_2 has only one possible pipeline configuration w_2.

Bit strings	000	001	011	010	110	111	101	100
Subsets	$\{0\}$	$\{w_2\}$	$\{w_1', w_2\}$	$\{w_1'\}$	$\{w_1, w_1'\}$	$\{w_1, w_1', w_2\}$	$\{w_1, w_2\}$	$\{w_1\}$
$\|W_1\|$	0	0	1	1	2	2	1	1
$\|W_2\|$	0	1	1	0	0	1	1	0
Subgraph $\in G'$	Not valid	g_1	g_2	g_3	Not valid	Not valid	g_4	g_5

high-level manner on which path variants should be included in an autonomously created pipeline.

The application developer can define functional paths following the format of Unix configuration files in the proposed implementation. The configuration defines *vertices* that represent one or more pipeline components. Those vertices are implemented as single components or as functional stages. Functional stages are implemented with the GStreamer API to construct *bins*.

The proposed API allows application developers to specify how many path instances should be instantiated in parallel. In practice, the middleware controls the amount of possible functional paths with the Binary Reflected Gray Code (BRGC) algorithm [126]. Table 4.2 is an example of how this algorithm is applied to the set of all available paths in W. In this example, there are two functional paths involved, W_1 and W_2. The functional path W_1 has two possible configurations w_1, w_1', and the functional path W_2 has only one possible configuration w_2. The resulting set of subgraphs $G' = \{g_1, \ldots, g_n\}$ creates the variant search space, i.e., possible pipeline configurations.

In this paper, we also describe in detail what is contained in the modeled context (see Section 7.2.2). We define context as any information that characterizes the user surroundings, preferences, application running environment, or network conditions, which impacts the functional and non-functional requirements of an application. Thus, we argue that multimedia applications should take into consideration the ultimate source or sink in a multimedia pipeline, even if it is a human being. An abstraction to this argument is illustrated in Figure 7.5.

For autonomous decision-making, DAMPAT allows users to express weighted preferences of properties of the application, e.g. properties of components in a multimedia pipeline. Thus, users can provide their preferences at their level of knowledge of the internal configuration of the application. DAMPAT uses multi-dimensional utility functions to autonomously select the pipeline variant that provides the highest utility for a given context. The description of variant selection in Section 7.2.4 is brief due to space constraints in the publication of this paper. Thus, we include a more detailed explanation on the design and application of utility functions in the next section.

4.3.1 Multi-dimensional Utility Functions

DAMPAT uses multi-dimensional weighted utility functions as a means to specify the objectives that guide the adaptation logic. Utility functions allow users and developers of multimedia components to provide their preferences. For example, a user that prefers two-channels audio processing when using headphones, or a developer of an audio component that prefers (i.e., recommends) to configure the component to process six channels when possible. The process where the autonomic manager clearly defines the utility of a pipeline variant based on the preferences of the user and developer is often called preference elicitation [178]. Each pipeline variant is represented as a subgraph $g \in G'$, and each variant provides a (multi-dimensional weighted) utility.

Utility (τ) is the degree to which a particular pipeline variant has the potential to satisfy the user's needs. This utility is computed as a scalar mapping of the operational parameters for the component. τ is an element of T in the mapping.

Utility function (μ) is the mathematical relation such that each variation in the *context* ($x \in X$) of the application or the user (environment and preferences) is associated with a real number. Each component in a multimedia pipeline has a set of properties ($p \in P$), i.e., parameterization variability represented by $v.P$, $i.P$, $o.P$, $m.P$, and each property has an associated utility function (μ). The mapping of the utility function, which is denoted by Equation (4.1), is the rule, by which a uniquely defined element $\tau \in T$ is assigned to every element of $x \in X$, and to every element of $p \in P$.

$$\mu \colon X \times P \to T \tag{4.1}$$

The domain of X and P is the set of the real numbers $\mathbb{R}$, as Equations (4.2) and (4.3). The set of properties P is part of the pipeline description represented as a subgraph $g \in G'$. The set of properties in a pipeline is also represented as $g.P$.

$$X = \{x \in \mathbb{R} \mid x \in X\} \tag{4.2}$$

$$P = \{p \in \mathbb{R} \mid p \in P, P \in G'\} \tag{4.3}$$

Utility functions (μ), represented in Equation (4.4), are specific per multimedia component. Developers of components implement the utility functions. An example of such implementation is ut, which takes two arguments: the preferred property value $u.p \in X$ specified by the user u, and the corresponding property value $g.p$ provided by the component in the pipeline variant being evaluated. For more details about ut see Sections 7.2.4 and 8.2.4.

$$\tau = \mu(x) \tag{4.4}$$

The range of utility functions is normalized to result between zero (worse) and one (best), as Equation (4.5). Utility values of a variant under different contexts are unrelated; likewise, utility values of different variants under the same context are unrelated. Utility functions are evaluated at run time because the values of arguments depend on the input stream and context while using the application.

$$\mu = \{\mu(x) \mid \ 0 \leq \mu(x) \leq 1\} \tag{4.5}$$

Priorities (ε or we) represent the level of interest, preference, weight or rank of specific properties of a pipeline variant. Users provide priorities to express the relevance of, for example, media modalities, bitrate (quality) of video, or number of audio channels. A property with low priority means that this property does not help the configuration to satisfy the user preferences. A negative priority (represented by Not-a-Number (NaN)) indicates that this property is undesired, for example, a negative video modality means that no variant with video processing should be selected at all. The value of the priority is a real number normalized to be between zero (lowest priority), one (highest priority), or the NaN value to reject the entire pipeline that is characterized by an unwanted property (negative priority).

Weighted utility function (σ) is a function of two variables, priority, i.e., weight (ε), and utility (τ), as Equation (4.6). The mapping of σ is denoted by Equation (4.7), by which a uniquely defined weighted utility element $h \in H$ is assigned to every prioritized (weighted) utility element $\theta \in \Theta$. θ represents the function that takes two arguments (priority and utility). The domain of Θ is $[0, 1]$, as Equation (4.8), where $\mathbb{R}^2$ represents a two-dimensional variable space. The co-domain (or range) of H is also $[0, 1]$ as Equation (4.9). The weighted utility function $\sigma(\varepsilon, \tau)$ is the scalar product of the weight (ε), and the utility (τ), as in Equation (4.10).

$$h = \sigma(\varepsilon, \tau) \tag{4.6}$$

$$\sigma \colon \Theta \to H \tag{4.7}$$

$$\Theta = \{(\varepsilon, \tau) \in \mathbb{R}^2 \mid 0 \leq \varepsilon \leq 1, 0 \leq \tau \leq 1\} \tag{4.8}$$

$$\sigma = \{\sigma(\varepsilon, \tau) \mid 0 \leq \sigma(\varepsilon, \tau) \leq 1\} \tag{4.9}$$

$$\sigma(\varepsilon, \tau) = \varepsilon \cdot \tau \tag{4.10}$$

Multi-dimensional weighted utility (Υ) is the sum of the weighted utilities H of all components in a pipeline. That is, the multi-dimensional weighted utility of one pipeline ($\upsilon \in \Lambda$) is equal to the output of the function Υ, as Equations (4.11) and (4.12), where the argument H is the set of weighted utility functions h, as Equation (4.6). Equation (4.13) is the mapping of Υ, by which a uniquely defined $\upsilon \in \Lambda$ is assigned to every element $h \in H$. The domain and co-domain of Υ is $[0, 1]$, as Equations (4.14) and (4.15), where n in Equation (4.14) is the *n-dimensional space* of independent variables to compute the overall utility of a pipeline.

$$\upsilon = \Upsilon(H) \tag{4.11}$$

$$\Upsilon(H) = \sum_{j=1}^{l} h \tag{4.12}$$

$$\Upsilon \colon H \to \Lambda \tag{4.13}$$

Table 4.3: Scenarios for assignment of utilities T of properties $\{p\}_{j=1}^{l=3}$ for different contexts $\{x\}_{j=1}^{l=3}$

Context $\{x\}$	Video (p_1)	Audio (p_2)	Text (p_3)
1	0.8	1	1
2	0	1	0

$$H = \{h \in \mathbb{R}^n \mid h \in H, 0 \leq h \leq 1\} \tag{4.14}$$

$$\Upsilon = \{\Upsilon(H) \mid 0 \leq \Upsilon(H) \leq 1\} \tag{4.15}$$

4.3.2 Examples of Multi-dimensional Utility of Pipeline Variant

Tables 4.3 to 4.5 show some scenarios to exemplify the definition in Section 4.3.1. Table 4.5 uses the values in Tables 4.3 and 4.4. Table 4.3 shows the utility provided by different properties in one pipeline variant for different situations (contexts). p_1 is the property of video modality, p_2 is the property of audio modality, and p_3 is the property of text modality.

Table 4.4 shows how weights are normalized between zero and one. By default, the weight of all properties sum 1, and are equally important, as shown in Scenario 1. If the user adjusts one weight, the autonomic manager ensures the sum of the weights is always equal to 1. The difference adjusted by the user is added or subtracted among all other weights. In this way, the manager preserves the relation to previous adjustments of importance. In Scenario 2, if the user wants to assign 0.70 as the priority for video, the manager normalizes the weights by calculating the difference ($0.7 - 0.33 = 0.37$) and dividing it by the remaining weights, then the result is subtracted from ε_2 and ε_3. Scenario 3 exemplifies the negation of modalities, where the user does not want the video modality. Note that the negation of a modality is different than assigning a weight equal to 0 as in Scenario 4. A weight with value 0 means that a given property will not be taken into account when calculating the overall multi-dimensional utility.

Table 4.5 shows some instances of the overall multi-dimensional weighted utility ($\upsilon \in \Lambda$) for one pipeline in a given context (as in Table 4.3) and priority (as in Table 4.4). Combination 1 shows the equivalence when assigning the same weight to all utilities. Combination 2 shows how the overall utility changes when

Table 4.4: Scenarios for automatic assignment of priorities $\{\varepsilon\}_{j=1}^{l=3}$.

Scenario	Description	ε_1	ε_2	ε_3	$\sum^{l=3} \varepsilon_{j=1}$
1	No priorities specified	0.33	0.33	0.33	1.00
2	Prefers video	0.70	0.15	0.15	1.00
3	NaN for unwanted video	NaN	0.5	0.5	1.00
4	Do not care about video	0	0.5	0.5	1.00

the user adjusts the priorities. Combinations 3 and 4 show the difference between assigning a NaN or 0 value as the weight for a property. Combination 5 shows the utility of a pipeline that does not contain (process) an unwanted property.

Table 4.5: Overall multi-dimensional weighted utility of one pipeline for a given context (in Table 4.3) and priority (in Table 4.4).

Combination	Utilities in context scenario	Priority (weight) scenario	v
1	1	1	0.924
2	1	2	0.86
3	1	3	NaN
4	1	4	1
5	2	3	0.50

4.4 P3 – DAMPAT: Dynamic Adaptation of Multimedia Presentations in Application Mobility

Authors: Francisco Javier Velázquez-García and Frank Eliassen

Authors' Contributions: Velázquez-García did the design, prototype implementation, evaluation and writing of this paper. Eliassen supervised the work for this publication. A major contribution from Eliassen was his advise to rely on Monitor, Analyze, Plan, and Execute (MAPE)-Knowledge (K) model, and Dynamic Software Product Lines (DSPLs) as research base.

Reference in Bibliography: [191]

Included in: Chapter 8

Address or Attends to:

- Research Questions 1 and 2 (stated in Section 1.2).
- Concerns in Monitor, Analyse, Plan and Execute phases on how to address variability of pipelines, how to detach mobile applications from host devices, how to adapt multimedia presentations, how to make autonomous decisions on pipeline configurations, and how to address scalability issues (see Table 4.1).
- API to be used at design time provided as a framework, and adaptation at load and run time handled by the middleware (see Table 4.1).
- Mathematical analysis and prototyping[4], as middleware (see Figure 4.1).

Summary and Thesis Relevance

For multimedia applications to adhere to the ubiquitous computing paradigm, they must be able to self-adapt to the receiving device. For this adaptation to happen autonomously, we argue that the application (or services used by the application) must make autonomous decisions based not only on the software and hardware in the receiving device, but also on the user preferences and users' physical environment. Our initial middleware designed in [96] (summarized in Section 4.7, and included in Chapter 11) considers the components needed to offer mobility services for process migration (i.e., mobile applications), however, it does not explain how the middleware performs autonomous decision-making.

In this paper, we present how to separate the concerns to tackle the complexity of autonomous self-adaptive mobile applications by following the MAPE-K autonomic adaptation loop model. The proposed solution based on this model is referred to as DAMPAT. DAMPAT follows the DSPL engineering approach. DAMPAT aims to provide the services needed by the middleware to support autonomous parameter setting (i.e., parameterization variability), component replacement (i.e., compositional variability), and component redeployment (e.g. socket migration).

Multimedia content is presented in a collection of multimedia streams. The specific configuration of the stream collection is called multimedia presentation. To adapt to a very large number of heterogeneous multimedia devices, we argue

[4]Source code available on `https://gitlab.com/francisv/gstreamer-devel-tests`, `https://gitlab.com/francisv/gstreamer-devel-tests-2`, and `https://gitlab.com/francisv/gstreamer-devel-tests-3`

that mobile multimedia applications should self-configure. In this way, applications can process the streams according to the specific I/O interfaces of the host device in use. Our approach to achieve this adaptation is to adapt the multimedia pipeline that processes the multimedia streams.

The design and implementation of capability negotiation mechanisms for multimedia pipelines is a complex task. Therefore, we investigated state-of-the-art multimedia frameworks including GStreamer [86], VLC[5], Qt[6], and FFmpeg[7]. Based on easiness, documentation, implemented components to build pipelines with different characteristics, I/O support in heterogeneous devices, distributed pipelines support (as in [116]), cross-platform support, and framework redistribution size, we decided to leverage the mechanisms of GStreamer. In addition, GStreamer is written in C, which makes it a good choice as research base due to language transparency when implementing: (1) graph routines in GNU Linear Programming Kit [134], (2) optimization to the decision-making algorithm in linear or mix integer linear programming (MILP) in GNU MathProg [136, 135], or (3) reformulation of the decision-making algorithm into the classical satisfiability problem (SAT) in [133].

The autonomic creation of the variability search space of possible multimedia pipeline configurations has in principle an exponential growth due to: (1) the uncontrollable number of available pipeline components, i.e., compositional variability, and (2) the uncontrollable number of parameterization possibilities per component, i.e., parameterization variability. DAMPAT mitigates the exponential growth by applying architectural constraints; it reduces compositional variability in functional stages (see Section 8.2.2), and control path combinations based on modalities (see Section 8.2.3). Developers of applications can adjust these constraints by describing the stages at their level of expertise, e.g. a developer can represent a *pre-processing* stage, which will match the metadata descriptors: *protocol handler, parser, demuxer* and *decoder*. Then, if components of the neighboring stages are compatible, they are linked (see Table 8.1).

Evaluation of this paper shows that the time to create an adaptation plan can be in the order of a few seconds. This time overhead is introduced at load time (when the application is loaded in the host device, or when the components in the device are updated). In the case where the user starts the application, we expect the user will not notice it, because a multithreading or multitasking application

[5]`https://www.videolan.org/`
[6]`https://www.qt.io/`
[7]`https://ffmpeg.org/`

would allow the user to perform other tasks while the Plan phase is executed. However, the time overhead can become annoying if the Plan phase is executed as the adaptation is required during a multimedia session. Therefore, the algorithms involved in the creation of the adaptation must be improved, or the adaptation plan must be created before adaptation is needed. We address the first alternative in [192] (summarized in Section 4.3, and included in Chapter 7).

For the second alternative (to create the variability search space on beforehand) we propose two approaches. First, the autonomic manager can send the metadata of the input stream to the collaborating devices where the application can be potentially moved, so DAMPAT can create the search space before adaptation is needed. Second, the autonomic manager can perform the planning phase entirely on a model of pipelines, such as in models@runtime [22]; for this, the designer of the model should abstract the capability negotiation mechanisms needed for autonomous configuration of the application, especially for multimedia pipelines. The work for these approaches, however, is left as future work.

GStreamer uses the GLib [184] library, and GLib uses the GIO library [183]. Thus, in order to transparently use the mechanisms for socket migration [194] (proposed solutions for Research Question 3), or data sharing [95] (proposed solutions for Research Question 4), we must integrate them in the GIO library. This integration is left as future work.

4.5 P4 – SOCKMAN: Socket Migration for Multimedia Applications

Authors: Francisco Javier Velázquez-García, Håvard Stigen Andersen, Hans Vatne Hansen, Vera Goebel and Thomas Plagemann

Authors' Contributions: Velázquez-García co-supervised the master thesis [9], which was the starting point for this paper. Velázquez-García wrote the version not accepted at the Middleware Conference 2012. For this submission, Velázquez-García, in collaboration with the co-authors, improved the content of the thesis mainly by studying the results thoroughly, adding new evaluation metrics and a new evaluation scenario, adding related work, and presenting results in a clearer manner. Both co-authors, Velázquez-García and Hansen, rewrote the not accepted version of the paper to achieve greater clarity. The paper was published at ConTel 2013 conference, and it was presented by Velázquez-García. Goebel and Plagemann co-supervised the work for the paper by discussing the arguments and structure of the paper.

Reference in Bibliography: [194]

Included in: Chapter 9

Address or Attends to:

- Research Questions 2 and 3 (stated in Section 1.2).
- Concerns in Execute phase on how to detach mobile applications from host devices, and how to provide IPC mechanisms for mobile applications with Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) connections.
- API to be used at design time provided as a framework, and adaptation at load and run time handled by the middleware (see Table 4.1).
- Prototype[8] as middleware (see Figure 4.1).

Summary and Thesis Relevance

In order to enable distributed components to communicate in a location-independent manner, we identify the following requirements: low connection handover time, sufficient throughput for the targeted application domain, portability, no modifications to the communications protocols used in the Internet, and no servers to manage connections handover. To the best of our knowledge, none of the related work (detailed in Section 3.6 and Section 9.5) meets all the identified requirements.

As part of the proposed middleware, we decided to provide the service for endpoint mobility at the transport protocol layer, because it allows to meet the requirements above mentioned, and it obeys the end-to-end principle [165] in system design. The design of the DARPA Internet standard protocols implemented in Berkeley Unix sockets, i.e., TCP and UDP, tightly couple the device and network identity with a 5-tuple of Internet Protocol (IP) address, port, and protocol at each endpoint of the connection. This 5-tuple represents the major challenge when addressing the mobility of endpoint connections at the transport layer.

[8]Source code available upon request at `https://gitlab.com/francisv/sockman`.

To address this challenge, we design, implement and evaluate a service called SOCKMAN, which reconfigures the 5-tuple of sockets that are moved to another device. The migrated sockets preserve the transport protocol state, and their mobility is hidden behind a proxy that tunnels entire IP packets through the proxy. As a result, applications using the services from SOCKMAN can interact with legacy applications.

The modification of standard implementations at the transport protocol layer for transparent use of legacy applications is an approach used by production-quality solutions such as in *SuperSockets* of Dolphin [58]. Thus, SOCKMAN represents an independent contribution to mobile networking and host mobility.

4.6 P5 – Efficient Data Sharing for Multi-device Multimedia Applications

Authors: Hans Vatne Hansen, Francisco Javier Velázquez-García, Vera Goebel, Thomas Plagemann

Authors' Contributions: Velázquez-García participated in discussions throughout all the process of the paper. Velázquez-García improved the clarity of the paper in the slides he made for the Middleware 2012 conference; he presented the paper. Hansen designed, implemented, evaluated and wrote the paper. Goebel and Plagemann co-supervised the work throughout the paper.

Reference in Bibliography: [95]

Included in: Section 4.6

Address or Attends to:

- Research Questions 2 and 4 (stated in Section 1.2).
- Concerns in Execute phase on how to detach mobile applications from host devices, and how to share data over Internet among distributed components.
- API to be used at design time provided as a framework, and adaptation at load and run time handled by the middleware (see Table 4.1).

- Prototype[9] as middleware (see Figure 4.1).

Summary and Thesis Relevance

In this paper, we investigate how distributed components of multimedia applications can share data over the Internet in an efficient manner. The presence of multiple consumers from one content source can reduce the throughput and available bandwidth in the source device. For example, think on a video conference session that moves (and duplicates) the video processing component to several other devices. Multimedia data distribution in such scenarios should be efficient.

Our solution provides mechanisms to distribute and replicate data segments in a transparent manner by providing location independent labels that identify multimedia content. The proposed packet layout to transfer data segments (illustrated in Figure 10.5) masks and hide heterogeneity of data representation in different architectures. Hence, the middleware provides transparent data access. Location and data transparency are necessary for the distribution of components of fine-grained multimedia applications.

The middleware implements mechanisms for data propagation, and it utilizes latency-optimized trees to distribute CPU and bandwidth usage among participants. For this purpose, we propose a publish-subscribe (PUB/SUB) [145] solution that leverages the mechanisms of two systems: (1) the Portable Operating System Interface (POSIX) shared memory API [182, 115], and (2) the coordination mechanisms using D-Bus [154].

The evaluation from the implemented prototype shows that the middleware selects the most efficient distribution path from producer to consumer. As a result, the local performance avoids the saturation of the data sources and access to memory is close to regular memory speed.

4.7 P6 – Migration of Fine-grained Multimedia Applications

Authors: Hans Vatne Hansen, Francisco Javier Velázquez-García, Vera Goebel, Ellen Munthe-Kaas, Thomas Plagemann

[9]Source code available upon request at `https://gitlab.com/francisv/edsmma`.

Authors' Contributions: Velázquez-García, in collaboration with the co-authors, designed the component-based TRAMP architecture. In particular, Velázquez-García proposed the modules: signaling, policies, and connection handover. The connection handover solution and results presented in this extended abstract come from [194] (summarized in Section 4.5 and included in Chapter 9). Velázquez-García and Hansen co-wrote the entire extended abstract. Velázquez-García, with feedback from the co-authors, made the poster of the extended abstract, and presented it at the Middleware Conference 2012. Hansen proposed the module: efficient data sharing memory. Goebel, Munthe-Kaas, and Plagemann supervised the work of this paper.

Reference in Bibliography: [96]

Included in: Chapter 11

Address or Attends to:

- Research Questions 1, 2, 3 and 4 (stated in Section 1.2).
- Concerns in Execute phase on how to detach mobile applications from host devices, how to provide IPC mechanisms for mobile applications with TCP and UDP connections, how to share data over Internet among distributed components, and how to reduce the hassle of PIM when using multiple devices.
- API to be used at design time provided as a framework, and adaptation at load and run time handled by the middleware (see Table 4.1).
- Prototype[10] as middleware (see Figure 4.1).

Summary and Thesis Relevance

This paper reports the initial research questions identified in this thesis. The research questions are mapped to the goals of eight components (parts of the middleware): (1) create device communities (addressed in [94]), (2) migrate processes, (3) perform signalling for process migration (addressed in [10]), (4) achieve transparent connection handover (addressed in [194]), (5) provide efficient data sharing

[10]Source code available upon request at `https://gitlab.com/francisv/edsmma` and `https://gitlab.com/francisv/sockman`.

(addressed in [95]), (6) describe and discover application components within device communities (addressed in [10]), (7) implement policies (addressed in [191]), and (8) aid users in configuring devices (addressed in [191]).

In an earlier stage of this thesis, we proposed the architecture of a middleware solution called TRAMP. The focus at this stage was to provide services in the less intrusive manner, and provide process migration services for legacy applications without modifications. To achieve process migration, we investigated the Java Virtual Machine (JVM), in particular, the Dalvik JVM [144] (discontinued since 2014), Maxine [200], Jikes RVM [173, 158, 159, 157, 38, 5, 66], and the Java framework OSGi [39, 205, 62]. However, after revisiting the Informational phase several times, we decided not to pursue the use of JVM for three main reasons. (1) Many multimedia-capable devices do not support Java, such as embedded devices. (2) JVMs use the least-common-denominator approach for exposing user interfaces, i.e., JVMs do not recognize and utilize non-standard device-specific peripherals. (3) Java is a proprietary language specification that is susceptible to copyright law infringement. Instead, we opted to continue the research of the middleware using the C programming language, which is light-weight and portable (despite false portability assumptions known as *vaxocentrism* [185]) to a large number of multimedia-capable devices. This approach is closer to *code mobility* than to process migration. Thus, the design and implementation of a JIT C compiler in the middleware are needed. The work to realize code mobility is, however, future research.

Further research after this paper brought additional research questions on how to provide services to application developers for the adaptation of multimedia content at run time, and how to achieve autonomous decision-making. Thus we had to make a clearer separation of concerns between the framework and APIs offered to application developers, and a unified autonomous control loop implementation in the middleware. After revisiting the informational phase (described in Section 1.4.1) several times, we proposed an evolution of TRAMP that follows the MAPE-K model [113].

4.8 Related Master Theses

This section briefly summarizes five master theses whose research questions were identified as part of this PhD thesis. These theses are supportive work that attends to the research questions stated in Section 1.2. The theses are presented in chronological order.

4.8.1 M1 – A Real-Time Video Retargeting Plugin for GStreamer

Authors: Haakon Wilhelm Ravik

Supervisors: Francisco Javier Velázquez-García and Thomas Plagemann

Reference in Bibliography: [161]

Address or Attends to:

- Research Question 1 (stated in Section 1.2).
- Concerns in Plan and Execute phase on how to adapt multimedia presentations.
- Adaptation at run time handled by the middleware (see Table 4.1).
- Prototype[11] as middleware (see Figure 4.1).

In this PhD thesis, we leverage the GStreamer multimedia framework because it provides an API to develop components for multimedia pipelines. The code base of GStreamer already contains a large number of pipeline components that provide different adaptation types (see Figure 3.1 for an overview of types of adaptation of multimedia data), and it provides mechanisms for building multimedia pipelines. However, this code base does not contain any retargeter. Therefore, Ravik designed, implemented, and evaluated a video retargeter (SeamCrop) for adaptation of Video on Demand (VoD) services as a GStreamer component. The availability of this component in the initial search space of components in a host device allows mobile applications to adapt the presentation of videos in small displays.

4.8.2 M2 – Negotiation and Data Transfer for Application Mobility

Authors: Marko Andic

Supervisors: Francisco Javier Velázquez-García and Thomas Plagemann

[11] Source code available upon request at `https://gitlab.com/francisv/GstSeamCrop`.

Reference in Bibliography: [10]

Address or Attends to:

- Research Questions 1 and 2 (stated in Section 1.2).
- Concerns in Execute phase on how to detach the application from host device, and how to reduce the hassle of PIM when using multiple devices.
- Capability negotiation at run time handled by the middleware (see Table 4.1).
- Prototype[12] as middleware (see Figure 4.1).

When several applications are running concurrently in one device or in the collaborating devices, the applications are competing for the available resources, and thus adaptation should be coordinated. The protocol designed, implemented, and evaluated by Andic allows DAMPAT to exchange data for decision-making on whether an application can be moved to a remote device or not. The proposed protocol eases the implementation of policies to evaluate whether the mobility of the application is worth it or not. This work also handles the data transfer of static and dynamic data of applications (or processes).

The protocol is also useful in such use cases to select the computer where a multimedia pipeline can perform best. For example, if the pipeline is described to use a special math co-processor, GPU or supercomputer, the protocol can select the most appropriate target computer. As a conclusion of this master thesis, we argue that negotiation protocols based on Session Initiation Protocol (SIP) are suitable for fine-grained mobile multimedia applications that adhere to the ubiquitous paradigm.

4.8.3 M3 – Adaptation trigger mechanism

Authors: Goran Karabeg

Supervisors: Francisco Javier Velázquez-García and Thomas Plagemann

Reference in Bibliography: [111]

Address or Attends to:

[12]Source code available upon request at `https://gitlab.com/francisv/negotiation-protocol`.

- Research Question 1 (stated in Section 1.2).
- Concerns in Analyse, Plan, and Execute phase on how to adapt multimedia presentations.
- Capability negotiation at run time handled by the middleware (see Table 4.1).
- Prototype[13] as middleware (see Figure 4.1).

Mobile multimedia applications require mechanisms to trigger adaptation of multimedia presentations not only when the application moves to a heterogeneous device, but also when the context change (while running the application) in the same device. The mechanisms to trigger adaptation of multimedia presentations in the same device were designed, implemented, and evaluated in the master thesis by Karabeg. The work of this thesis is part of the Analyse phase (in the MAPE model), and is responsible for identifying when the application configuration does not support the current safety predicate anymore (explained in Section 7.2.3).

4.8.4 M4 – Component-based multimedia application for fine-grained migration

Authors: Tomas Gryczon

Supervisors: Francisco Javier Velázquez-García, Hans Vatne Hansen, and Thomas Plagemann

Reference in Bibliography: [83]

Address or Attends to:

- Research Questions 1 and 2 (stated in Section 1.2).
- Concerns in Execute phase on how to detach applications from devices in a fine-grained manner.
- Prototype[14] at application layer (see Figure 4.1).

[13]Source code available upon request at `https://gitlab.com/francisv/trigger-mechanism`.

[14]Source code available upon request at `https://gitlab.com/francisv/component-based-mmm-app`.

In this PhD thesis, we argue that fine-grained application mobility is needed to take advantage of the heterogeneous multimedia-capable devices in the ubiquitous computing paradigm. Thus, we have argued that application developers should split their applications into components that can be executed in different devices, i.e., components implemented to run as independent processes in the operating system. Gryczon designed, implemented and evaluated a multimedia application following this separation of components into processes to show the suitability of this approach.

4.8.5 M5 – User Space Socket Migration for Mobile Applications

Authors: Håvard Stigen Andersen

Supervisors: Francisco Javier Velázquez-García, Hans Vatne Hansen, Vera Goebel, and Thomas Plagemann

Reference in Bibliography: [9]

Address or Attends to:

- Research Question 3 (stated in Section 1.2).
- Concerns in Execute phase on how to provide IPC mechanisms for mobile applications with TCP and UDP connections.
- API to be used at design time provided as a framework, and adaptation at load and run time handled by the middleware (see Table 4.1).
- Prototype[15] as middleware (see Figure 4.1).

In this PhD thesis, we argue that mobility at application-level, i.e., application mobility, is part of the solution to realize the vision of ubiquitous computing. One part of applications are their sockets; hence sockets must move together with their applications. Andersen designed, implemented, and evaluated mechanisms to hide the mobility of TCP and UDP sockets between devices. In this way, mobile applications can interact with legacy applications.

[15]Source code available upon request at `https://gitlab.com/francisv/socket-migration`.

4.9 Summary

In this chapter, we summarized the already published research work that has been done in the direction of the aim of this thesis: ease the development of multimedia applications that adhere to the ubiquitous computing paradigm. The author's contribution per work has also been stated.

We have illustrated in Figure 4.2 how the MAPE autonomous adaptation loop model separates the concerns of the proposed middleware. Table 4.1 shows the relation between the research questions, concerns, phases (in the MAPE model), framework (concerns at design time), and middleware (concerns at load time and run time). Figure 4.1 presents the relation of research work as building blocks in the implemented middleware.

Chapter 5

Conclusions

We summarize in Section 5.1 what we have learned from the work in this thesis and the significance of the results. The core of the thesis is the proposed middleware and Application Program Interfaces (APIs) that provides new modes of action for others to achieve fine-grained application mobility that adheres to the ubiquitous computing paradigm. Application mobility is an efficient and scalable way to enable users to take advantage of the dynamically changing set of surrounding devices during a multimedia session; this statement is supported by the results summarized in Chapter 4. We provide a critical review and open issues in Section 5.2, future work Section 5.3, and future research in Section 5.4.

5.1 Summary of Main Contributions

We have proposed a novel middleware and framework for fine-grained mobile multimedia applications. The middleware is designed as an autonomic adaptation loop and chooses the variant that provides the highest utility for the user at load and run time. The framework provides an API that simplifies software development by allowing the developer to pass high-level goals of services as arguments. The framework allows developers to capture (at design time) the variability of user physical environment, user preferences, application runtime environment, and network conditions. The implemented mechanisms encapsulated in the APIs give access to resources in a location-independent and seamless manner. Applications using the proposed API and middleware acquire the self-adaptive, self-configuration, self-optimization, and self-awareness properties.

To achieve adaptation of multimedia presentations in the ever-growing diversity of heterogeneous multimedia devices in a timely manner, we propose to adapt multimedia pipelines. We contribute with a series of architectural constraints to address complex and NP-hard problems introduced by the inherent unexpected variability in ubiquitous computing. By complex problems, we refer to the difficulty level of detail to ensure multimedia pipelines are correctly built and reconfigured in an autonomic manner, while media streams are processed efficiently and in synchrony. By NP-hard problems we refer to the combinatorial explosion in the variability of multimedia components and their parameterization. Evaluation of prototypes, implemented in the GStreamer multimedia framework, demonstrates that the reconfiguration of multimedia pipelines approach allows a seamless use of different adaptation types that may even be unknown at design time.

Application mobility implies the mobility of connection endpoints. However, the widely adopted protocols Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) in Internet do not provide the needed mobility. To address this problem, we contribute with a proxy-based socket migration service that allows efficient continuity of TCP and UDP endpoint connections. This service meets communication requirements of multimedia applications.

In application mobility, components of multimedia applications that access data from the local memory should also be able to access it from the network at a fast enough speed to meet strict multimedia deadlines. For that, we contribute with the creation of an inter-process communication (IPC) service that allows efficient data sharing across devices. The service creates an overlay network that connects the devices hosting the distributed application, and act as an application collaborator, by implementing a publish-subscribe (PUB/SUB) service, to share data. As a result, fine-grained applications running in different devices are able not only to process multimedia content in a distributed manner, but also to support associated computing and communication tasks.

In a greater context, the contributions from this thesis are also relevant to applications that seek to increase the availability of resources. For example, applications in cyber foraging [15] seeking to increase the capabilities of resource-limited mobile devices, or applications in cloud computing.

5.2 Critical Review and Open Issues

To ease the development of multimedia fine-grained mobile applications, we have implemented, designed and evaluated services that are to be used by application developers. Admittedly, the implementation is of prototype quality, meaning that it requires more implementation to catch all types of errors when probing invalid pipelines. A more robust error handling will ensure that the created variability search space will contain only those pipelines that can process streams without errors.

Due to time constraints in the implementation, the prototypes to verify each proposed service, calls only the functions of the library in question. For example, the prototypes involving multimedia pipelines (see Section 4.2) do not call the functions from the proposed APIs for socket migration (see Section 4.5) or data sharing (see Section 4.6). That is, the use of all services by one application is unverified.

For a transparent use of the API, i.e., be compliant with legacy software, the current implementation of the connection handover service (see Section 4.5) must not only resemble the Portable Operating System Interface (POSIX).1-2008 specification, but must conform to it. An alternative is to integrate the current implementation to the GIO library [183] (used by GLib [184]).

In general, the source code should be improved to integrate the services in one middleware in a plug-and-play fashion. In this way, the services can become as mobile as the applications running on it. Documentation should be added and the source code not yet public should be published, so other academics and developers can use, study, share and improve it.

5.3 Future Work

We have presented in Section 7.2.1 the data model for capturing the variability of context[1]. However, the engineering work to implement the actual mechanisms to collect, aggregate and analyze the data, is still to be done. If these mechanisms are not designed and implemented efficiently, there is a risk of overloading the system. For this task, we suggest to research monitoring tools as first-class citizen as in [33, 29]. Once monitoring data is collected, an important engineering question is how to make this information available to the applications in a way that satisfies best practices of software engineering.

The implementation of the socket migration service, called SOCKMAN (summarized in Section 4.5 and included in Chapter 9), can be extended by adding, for

[1]Implementation of context models as MySql relational database models available on `https://gitlab.com/francisv/dampat`.

example, support for window scaling. Also, the current evaluation of state preservation in TCP connections is valid, but its performance evaluation cover just in a few test cases. Further scenarios should be evaluated as suggested in [4].

Much work, e.g. [64, 7, 131, 8], has been done on autonomous decision making based on Quality of Service (QoS). In this thesis, autonomous decision-making is made by calculating the utility of multimedia pipeline variants, and selecting the variant with the highest utility in a given context. If the utility functions, developed by the developers of the components, take into consideration the performance properties of QoS, then the utility provided by the component should affect the overall multi-dimensional utility function, defined in Equation (4.12). The implementation and evaluation of such behavior are, however, left as future work.

5.4 Future Research

Checkpointing mechanisms typically used in process migration systems [140] are needed to achieve code mobility (as in Section 2.2); these mechanisms handle the static and dynamic state of processes in the operating system. Developers of new applications can explicitly define the static and dynamic parts, but it might not be trivial to identify what comprises the static and dynamic parts in legacy applications, especially if the application was written by someone else. Thus, we suggest research in how stub-generation tools can generate application-specific interfaces that help developers to identify parts of legacy applications that have to be updated to become migration-aware. For this, the source code of the legacy application in question has to be available. An example of this approach is in [14].

Applications that amplify the capabilities of resource-limited devices by offloading computation to a server are said to use the *cyber foraging* technique [15, 72]. Cyber foraging has two remain challenges [13]. First, it misses a compelling application. Second, setup and maintenance of the servers, i.e., *surrogates*. The mobility of multimedia pipelines and migration of endpoint Internet connections can help developers of cyber foraging applications to enable similar use cases to the ones described in Section 2.1 and Appendix B, but using surrogates. The second challenge is not present when developing mobile applications as described in this thesis, because the applications are moved to devices that the user would in principle trust, and the distributed data sharing solution (see Section 4.6) aggregates the CPU and bandwidth resources of the devices that the user trust. Research in how the proposed solutions in this thesis can help cyber foraging can bring insights on the trade-offs between using surrogates or moving applications to devices with the middleware pre-installed.

Authors in [206] explain how to model and parse conditional user preferences defined in semi-natural rules, and the authors claim that ontology-based quantitative models for this purpose are feasible. We would like to explore the benefits of this approach, instead of the direct user preference input of property and value tuples as in our context model.

Part II

Research Papers

Chapter 6

P1 – Dynamic Adaptation of Multimedia Presentations for Videoconferencing in Application Mobility

Authors: Francisco Javier Velázquez-García, Pål Halvorsen, Håkon Kvale Stensland, and Frank Eliassen

Published in: International Conference on Multimedia and Expo (ICME 2018)

Reference in Bibliography: [193]

Abstract: Application mobility is the paradigm where users can move their running applications to heterogeneous devices in a seamless manner. This mobility involves dynamic context changes of hardware, network resources, user environment, and user preferences. In order to continue multimedia processing under these context changes, applications need to adapt not only the collection of media streams, i.e., *multimedia presentation*, but also their internal configuration to work on different hardware. We present the performance analysis to adapt a videoconferencing prototype application in a proposed adaptation control loop to autonomously adapt multimedia pipelines. Results show that the time spent to create an adaptation plan and execute it is in the order of hundreds of milliseconds. The reconfiguration of pipelines, compared to building them from scratch, is approximately 1000 times faster when re-utilizing already instantiated hardware-dependent

components. Therefore, we conclude that the adaptation of multimedia pipelines is a feasible approach for multimedia applications that adhere to application mobility.

6.1 Introduction

Application mobility is a paradigm that impacts the means to produce or consume multimedia content when an application is moved into a different running environment [204]. For example, suppose a user is participating in a videoconferencing session using a mobile device while commuting. When the user arrives in her office, she continues the same session by moving the application to a dedicated office videoconferencing device with different I/O interfaces.

Proper support for application mobility with dynamic adaptation is very advantageous for users and developers of multimedia applications. Users can take advantage of different device capabilities as they become available in their environment, without interrupting ongoing multimedia sessions. Application developers can take advantage of already available mechanisms to ease the development of applications that are able to execute in, and move to, devices with characteristics that were unknown at design time, and that require adaptation beyond predefined profiles. Such mechanisms should provide internal reconfiguration due to different components in the receiving device, migration of connection end points, protocols for process migration, efficient distributed shared memory, and security under migration, among other.

In this paper, we address the dynamic adaptation of multimedia content in application mobility. In such mobility situations, two aspects of multimedia applications need to be adapted; multimedia content, composed of a collection of media streams, i.e., a *multimedia presentation*; and internal configuration to work on different hardware.

In videoconferencing use cases, each peer acts as a producer and consumer of multimedia content. Therefore, adaptation is required when producing or consuming content. We evaluate a videoconferencing prototype application that simulates not only the consumption of multimedia content as in [191], but also the production of multimedia content.

Multimedia content can be adapted into different multimedia presentations by different adaptation types, namely *fidelity*, *modality*, *content-based*, or *retargeting* adaptation. In order to adapt to the very large diversity of devices and situations while satisfying user preferences and QoS requirements, applying only one adaptation type is not enough. Yet, the more adaptation types are applied to some content, the more variants of it are created; and this situation can rapidly become a scaling and management issue.

Multimedia presentations are processed by sequentially connected components, a.k.a. *multimedia pipelines*. Depending on the needed adaptation type, streams in multimedia presentations can be adapted either by tuning the parameters of the components in the pipeline, changing the components themselves, or changing the topology of the pipeline. The mechanisms needed to manage multimedia pipelines have been addressed by different multimedia frameworks. However, to the best of our knowledge, GStreamer [86] is the only open source framework that actively maintains the mechanisms to create, manage, and dynamically reconfigure multimedia pipelines. Therefore, we have leveraged the mechanisms of GStreamer with a runtime adaptation control loop.

GStreamer includes an implementation for pipeline generation on player startup, but this cannot be used out-of-the-box in applications adhering to application mobility because; 1) pipelines are generated at startup with modality selection as only reconfiguration alternative during runtime, and 2) the pipeline generation is designed to consume multimedia content, not to transcode or produce it.

In order to autonomously create, and reconfigure multimedia pipelines, the combinatorial growth (of the ever growing number of pipeline components, and the large number of tunable parameters of pipeline components) has to be controlled. The contribution of this paper is the design, implementation, and performance analysis of the creation and execution of adaptive multimedia pipelines adhering to the mobility paradigm as part of a proposed adaptation control loop. We address the combinatorial growth by allowing developers to introduce design knowledge as architectural constraints. In comparison to the vanilla GStreamer framework [86], Infopipes [27], PLASMA [122] or Kurento [65], our approach enables the application developers and users to introduce high-level goals, without requiring deep knowledge of all components and their configuration details.

Multimedia applications following our approach are able to handle multiple modality streams, and they are able to autonomously adapt them either by tuning parameters of already instantiated components, changing components, or changing the topology of the application's multimedia pipeline. The measured adaptation time is up to hundreds of milliseconds when instantiating pipelines from scratch, and pipeline reconfiguration is about 1000 times faster when re-utilizing already instantiated hardware-dependent components.

The rest of the paper is organized as follows. In Section 6.2, we first explain the engineering approach, adaptation model, and multimedia pipeline model. Then, we describe how multimedia presentation variants (that constitute the variability search space) are created. Section 6.3 describes the main GStreamer mechanisms used to make the videoconferencing prototype, and their limitations for supporting application mobility. In Section 6.4, we evaluate the plan and execution phase of the proposed system, and discuss the scalability problems. Section 6.5 compares the results with related work. Finally, Section 6.6 concludes that reconfiguration of pipelines is a feasible approach to adapt multimedia presentations in application mobility, and it has two main advantages: avoids time overhead of hardware instantiation more than once, and mitigates the unpredictability of query mechanisms in GStreamer.

6.2 Design

The proposed system adopts the Dynamic Software Product Line (DSPL) engineering approach. In DSPL, designing a runtime adaptive system is considered to be a variability management problem, where variability of the system is captured at design time, and the *best* product variant is selected at runtime. *Best* is the variant that produces the highest utility according to the current contextual situation. In this paper, the utility function is out of scope, and we refer the interested reader to our previous work [191] for details.

To break down the concerns of the system, we follow the *Monitor, Analyze, Plan, and Execute (MAPE)-K* adaptation control loop [113]. In the MAPE model, the *Monitor* phase collects information from the sensors provided by the managed multimedia pipeline, and user's context, and preferences. The *Analyze* phase uses the data of the Monitor phase to assess the situation. When the Analyze phase detects that the utility provided by the current pipeline configuration is below a given threshold, it starts the *Plan* phase to generate an adaptation plan. The *Execute* phase applies the generated adaptation plan on the managed pipeline. *Knowledge* is created, and shared by all phases, and it holds information that impacts the production or consumption of multimedia presentations by the user or application. These phases constitute the context-aware *autonomic adaptation manager* of the system, which controls the *managed multimedia pipeline*. In this paper, we focus on the proposed Plan and Execute phases, and exemplify their use for a videoconferencing use case.

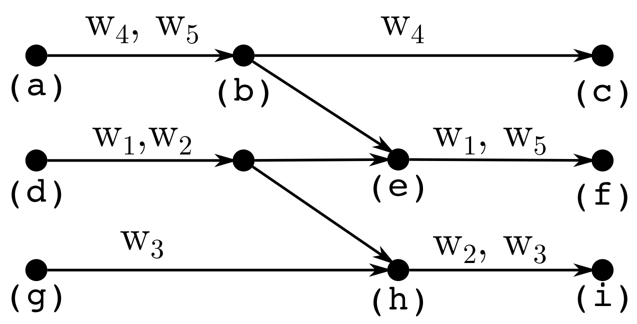


Figure 6.1: Abstraction of pipeline of videoconferencing peer A that produces and consumes audio and video modality.

6.2.1 Multimedia pipeline model

Multimedia presentations are processed by sequentially connected components, i.e., *multimedia pipelines*. Depending on the needed adaptation type, i.e., *fidelity*, *modality*, *content-based*, or *retargeting* adaptation, streams in multimedia presentations can be adapted either by: 1) tuning the parameters of the components in the pipeline, e.g., changing the `lowres` property of GStreamer H.265 decoder component `avdec_h265` to select which resolution to decode images; 2) changing the components themselves, e.g., replacing GStreamer components `vp9enc` with `x265enc`; or 3) changing the topology of the pipeline, e.g., removing components that process video (at the producer and consumer) when the user is driving.

Multimedia pipelines can be modeled as directed acyclic multigraphs $G = (V,E)$. In this abstraction, V is the set of vertices that represents the components in the pipeline, and E is the set of edges that represents a connection or pipe between the output, and input connectors of two pipeline components. As an example, Figure 6.1 is a graph abstraction of the pipeline that produces, and consumes content in: audio and video modalities, in peer A of a videoconferencing application. Peer A communicates with peer B (not shown in the figures). Figure 6.1 shows 5 abstractions of paths. The functionality of each path is as follows.

Path w_1 captures video from the webcam, and renders it in the display. Path w_2 captures video from the webcam, and sends it over the network. Path w_3 captures audio from the microphone, and sends it over the network. Path w_4 receives audio from the network card, and sends it to the audio card for reproduction. Path w_5 receives video from the network card, and renders it the display.

Suppose that peer B (communicating with peer A) has moved the application to a device without audio capabilities, and triggers pipeline reconfiguration in both

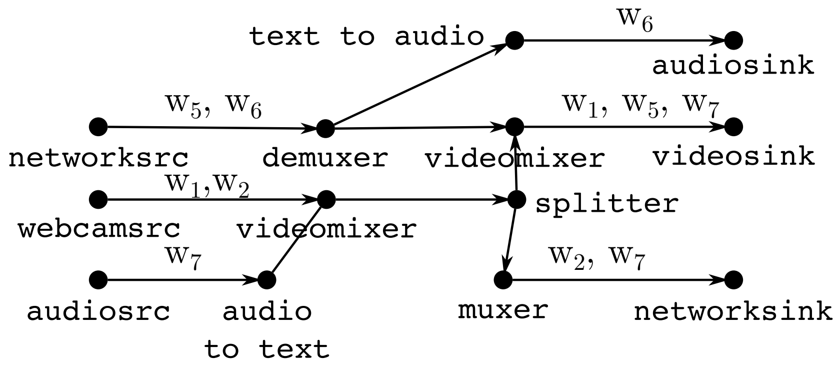


Figure 6.2: Abstraction of pipeline of videoconferencing peer A that has adapted to a peer without audio capabilities.

peers. Figure 6.2 shows the adapted pipeline in peer A to produce and consume text modality instead of audio. As a result, peers A and B transfer data containing video and text modalities, which in turns saves bandwidth. Note that, in this example, the user's I/O interfaces in peer A do not change. In Figure 6.2, path w_4 has adapted to path w_6, and path w_3 has adapted to path w_7. Paths w_6, and w_7 convert text modality to audio, and audio to text respectively.

Each needed component in a pipeline can have more than one candidate, which is referred to as *compositional variability*. For example, the components `v4l2src`, and `uvch264src` can be alternative candidates for capturing video from the web camera.

In a similar manner, every vertex has *parameterization variability* due to assignable property values of vertices ($v.P$), connectors ($i.P$, and $o.P$), and modalities ($m.P$). Compositional, and parameterization variability can create a rapid growth of complexity due to combinatorial explosion.

6.2.2 Plan Phase

In the Plan phase, the adaptation manager creates variants of *valid* multimedia pipelines, and selects the best one for a given context. By a valid multimedia pipeline, it is meant a pipeline with adequate configuration for the available resources, so that buffers arrive on time at the final sink. The Plan phase also reduces the variability growth by allowing the designer of the multimedia application to introduce architectural design knowledge, i.e., *architectural constraints* [77].

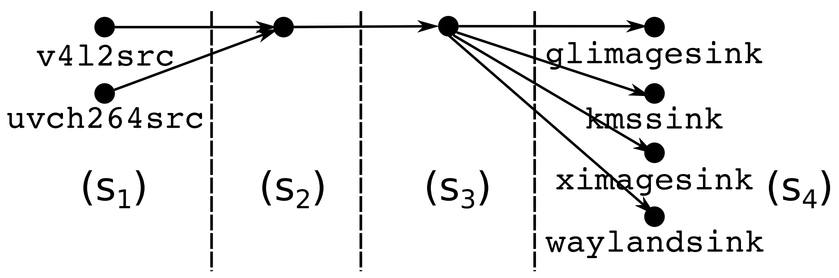


Figure 6.3: Example of functional stages $\{s\}_4$, for path w_1 (captures video from the webcam, and renders it in the display). In this example, w_1 has eight possible combinations.

Control of combinatorial growth due to compositional, and parameterization variability

We arrange the multigraph abstraction in a sequence of *functional stages* that defines the functionality of each processing step needed in a path. Application developers define functional stages to filter components by functionality, and the developers can specify the stages at different levels of accuracy as explained in our previous work in [191]. The stages act as architectural constraints per multimedia stream path to enforce directed graphs, and avoid unnecessary checks of connectors compatibility, which are most likely to fail.

As exemplified in Figure 6.3, a developer can provide the functional stage s_1 to group the components that capture video, and s_4 to group the components to render video. Stage s_2 is a specific component to fix the desired output of s_1, and s_3 does conversion of color space. These stages are part of the stages belonging to path w_1 that captures video from the webcam, and renders it in a X11 window.

Control path combinations

Due to the compositional variability in functional stages, multimedia streams may have a set of alternative configuration for one path. For example, Figure 6.3 shows the candidates `v4l2src`, or `uvch264src` for the path that renders the video captured by the camera. In order to restrict path combinations, we introduce an architectural constraint to limit the path combinations, where the upper bound of allowed path combinations is specified by the application developer.

In our previous work [191], we showed that when the developer decides to restrict path configurations to one in an application with three needed paths; e.g., video rendering, video transmission, and audio transmission, the combinatorial growth is reduced to the polynomial form.

To enforce the path combination constraint, the adaptation manager computes the Binary Reflected Gray Code (BRGC) algorithm (explained in in [191]). The output of the BRGC algorithm is a set of subgraphs $G' = \{g_1, \ldots, g_n\}$ that composes the variant search space. Each element $g \in G'$ represents a pipeline that can be instantiated in the Execution phase. g contains the description of the properties P of each vertex in g, the set of modalities M occurring in g, the properties P of each modality, and the set of edges E in G'. G' is part of the knowledge base of the system, and its elements are used as input for the utility function used in the decision making process.

6.2.3 Execution Phase

The task of the Execution phase is to safely introduce, remove, or re-configure components in the pipeline. Mechanisms to create, manage, and dynamically reconfigure multimedia pipelines include: connectors compatibility check, stream flow control for linking and unlinking connectors, stream flow control to handle delayed buffers in sinks due to limitations in local resources or bandwidth, pipeline state management, components instantiation, and memory allocation type check to avoid memory copying. For this purpose, we leverage the GStreamer mechanisms.

We assume states of components and pipelines are preserved when moving between devices by using check-pointing, store, and transfer state mechanisms at the stack level. In the case of changing components, state is preserved by reading the timestamps of the stream being processed. Sections 6.3.2, and 6.3.3 explain how a given plan is executed as creation or reconfiguration of a pipeline.

6.3 Implementation

In this section, we present the implementation of an application that contains multimedia pipeline components used for production, and consumption of multimedia content. In order to be able to adapt multimedia presentations processed even in embedded systems, such as multimedia systems in cars or airplanes, we address the instantiation and reconfiguration of multimedia pipelines at C programming language level. Since videoconferencing applications act as server, and client at the same time, we implement, and evaluate a videoconferencing prototype. The prototype leverages GStreamer `1.15 (GIT)`.

The prototype is designed for multimedia pipelines with one, two or three multimedia paths. Path 1 (w_1) captures video from the webcam, renders it in a X11 window. It has two functional stages: one for video capture, and another for video rendering. Path 2 (w_2) simulates video transmission by encoding the captured data, storing it in a file. It has one functional stage for video capture. Path 3 (w_5) simulates close captioning by displaying a clock overlay of the host in the X11 window. It has one functional stage for video rendering. Combination of candidate paths are restricted to 1.

6.3.1 Filter components per functional stage

The adaptation manager reads the metadata of components in order to match, and filter those components that are compatible with the metadata in functional stages, as describe in Section 6.2.2. In principle, compatibility checking can be done by checking the properties in the metadata registry. However, some components such as `opusdec`, and `vp9dec`, define their output connector based on the input stream *and* required output, because not all input streams may have the necessary metadata to help determine the output format. Therefore, either manual configuration, such as setting a `capsfilter` component with values `video/x-raw,format=(string)YUY2`, or the use of default values of components is needed for some pipelines.

6.3.2 Linking connectors

In order to check compatibility between connectors, the adaptation manager uses the GStreamer queries `query-caps` and `accept-caps` to check the processing capabilities of connectors. In [191], we have analyzed how the current approach of GStreamer, to register parameterization variability in components, can potentially introduce scalability issues in the autonomous creation of the variability search space. Therefore, we evaluate this behaviour in Section 6.4.

6.3.3 Dynamic reconfiguration

In the Execution phase, there is the choice for the application manager to either create the pipeline from scratch or reconfigure it. The pipeline is described in a graph g as explained in Section 6.2.2. One goal of the Execution phase is to reuse the already instantiated components when possible. For example, when the user of the videoconferencing application prefers close captioning instead of audio modality, the path for audio modality is removed, and the path for close captioning is added to the video rendering path.

In this prototype, we implement both alternatives; to build a pipeline from scratch, and to reconfigure a running pipeline. To build the pipeline from scratch, the adaptation manager execute the plan by doing the steps in Sections 6.3.1, and 6.3.2. To reconfigure a pipeline, we describe next the challenges of dynamic reconfiguration, and the solutions the adaptation manager implements.

The adaptation manager assumes multiple paths should be processed simultaneously, for example, to encode audio, and video in parallel. To ensure this behaviour, the adaptation manager creates a thread per path by inserting a `queue` component.

If a component is removed while it is processing a buffer, the thread processing the path can potentially enter in a deadlock state, because some other component in the path might indefinitely wait for the expected data to arrive. For this, the adaptation manager blocks the data flow in the preceding connector of the component that will be removed, and install a callback to be notified about the states of the data flow. The callback we use in GStreamer is `GST_PAD_PROBE_TYPE_IDLE`. After removing, and adding components, the adaptation manager synchronizes the state of all components to avoid deadlocks.

A typical race condition when reconfiguring pipelines occurs when a certain component in the pipeline waits for some timestamp or other specific data that was in the buffer or a just removed component. The adaptation manager handles this situation by flushing the buffers of the components to be removed.

When reconfiguring a pipeline, compatibility check is triggered. This process can be time consuming, or can require input from the user to define the format of the new configuration. To ease this situation, the application developer should define the preferred format of a stream by adding functional stages or `capsfilter` components as described in Sections 6.2.2 and 6.3.1.

6.4 Evaluation

In this section, we evaluate the scalability performance of the Plan, and Execution phase for the videoconferencing prototype that simulates the production and con-

Table 6.1: Experiments of time in Plan phase

Exp	Path	Funct. Stages	Path cand.	Num. comp.	Total queries	Repeated queries	Avg. (ms)	3rd Qu. (ms)
1	w_1	2	8	8	2984	736	10.2	10.5
2	w_5	1	2	9	832	86	0.1	0.1
3	w_2	2	8	8	1624	648	10.1	10.2

sumption of multimedia content. As a testbed, we use a computer that resembles hardware characteristics of commodity hardware. The computer is a MacBook Pro 7,1 with Intel Core 2 Duo CPU P8800 at 2.66GHz running the 64-bit Ubuntu 17.10 operating system. The initial search space has 1420 pipeline components.

A `start` timestamp is recorded right after initializing the application graphical interface, GStreamer library, setting up internal path lists, registering pipeline components, and loading standard plugins. After the pipeline is built, an `end` timestamp is recorded. The difference between `end`, and `start` is the response time to build one pipeline variant. For the experiments of pipeline reconfiguration, the `start`, and `end` timestamps are taken before and after reconfiguration is done.

We count the number of queries `query-caps` , and number of queries `accept-caps` in all experiments since they have been identified as scalability factors in [191]. We run a set of experiments starting from 10 to 100 repetitions to observe difference in response times, and 75% quantile ("3rd Qu." column in the Tables 6.1 and 6.2). Results from varying repetitions show differences in the order of nanoseconds, which we regard as negligible.

6.4.1 Plan phase

The key factors that influence the measurements of experiments for creating the search space are; the number of functional stages per path, number of components in the pipeline, and number of allowed path combinations. The values of these factors, and results of the experiments are summarized in Table 6.1. An approximation of the time to create the entire search space is the sum of the average for every candidate path, which results in 162.6 ms $(8 \times 10.2 + 10.1 \times 8 + 0.1 \times 2)$.

Although the number of components in experiments 1 and 3 are the same, the number of total and repeated queries vary. The response time of experiments 1 and 3 are very similar, and greater than in experiment 2 because most of the time was taken to instantiate the component for the video camera.

Table 6.2: Experiments of time in Execution phase

Exp	Path comb.	Exec. type	Prev. exp.	Num. comp.	Total queries	Repeated queries	Avg. (ms)	3rd Qu. (ms)
4	w_1	scratch	N/A	8	287	76	1.2	1.3
5	w_1, w_5	scratch	N/A	9	358	105	1.2	1.3
6	w_1, w_2	scratch	N/A	13	429	131	1.3	1.3
7	w_1, w_5, w_2	scratch	N/A	14	529	173	1.2	1.3
8	w_1, w_5	reconf	4	9	83	47	0.008	0.002
9	w_1	reconf	8	8	41	23	0.001	0.001
10	w_1, w_2	reconf	4	13	190	70	0.066	0.076
11	w_1	reconf	10	8	0	0	0.001	0.004

6.4.2 Execution phase

For the reconfiguration experiments, the number of components that are before, and after a reconfiguration point are also key factors. We identified a clear pattern of approximately 1 ms faster when reconfiguring a pipeline (instead of building it from scratch). We present only a selection of all combinations in Table 6.2 due to space limits. The values of the relevant factors, and results are summarized in Table 6.2.

Results in Table 6.2 show that reconfiguration of pipelines is approximately 1000 faster than creating the pipeline for scratch. This speed gain is mainly because hardware-dependent components do not have to be re-instantiated. The queries used to reconfigure pipelines continue to be unpredictable, mostly due to the different implementation of handlers for the `accept-caps` query in the pipeline components. Experiments 9 and 11 show that removing paths by reconfiguration can use 0 queries.

Results from reconfiguring pipelines in our prototype are clearly better than building pipelines from scratch. However, the implementation of more generic reconfiguration mechanisms in the Execution phase is more complex. Therefore, one should evaluate the trade-off between the benefit of the speed gain in reconfiguring pipelines against building pipelines from scratch, and the complexity associated to implement either approach for a given application domain.

6.5 Related work

The code base of GStreamer includes an implementation for pipeline generation on player startup, but this cannot be used out-of-the-box in applications adhering to application mobility because: pipelines are generated at application startup with modality selection as the only reconfiguration alternative during runtime; and the pipeline generation is designed to consume multimedia content, not to produce or transcode it.

Solutions for different architectures of infrastructures dealing with configuration or parameterization variability, as in the case of Infopipes [27] or PLASMA [122], will present an exponential growth, which is a NP-hard problem. Neither Infopipes nor PLASMA discusses how to limit this growth. We did not leverage Infopipes or PLASMA because their authors do not define or implement the needed mechanisms to create, manage, and dynamically reconfigure multimedia pipelines.

The Infopipes abstraction [119, 27] simplifies the task of building distributed streaming applications by providing basic elements such as pipes, filters, buffers, and pumps. Infopipes, as opposite to our work, triggers adaptation based on variations of resources of CPU and bandwidth only. Adaptation is achieved by adjusting the parameters of elements, but not by changing them or changing the path of the stream.

PLASMA [122] is a component-based framework for building multimedia applications. PLASMA relies on: a hierarchical compositional, similar to functional stages (Section 6.2.2); and a reconfiguration model, similar to the Execution phase (Section 6.2.3). PLASMA describes the mechanisms needed to build and reconfigure pipelines at runtime at a high level. However, they do not talk about the needed mechanisms to synchronize multiple streams, namely, clock synchronization, multi-threading management, and memory management. The authors do not describe how to specify multiple streams or media types, therefore, we regard their work valid for monomedia, not multimedia.

To implement dynamic reconfiguration of multimedia pipelines, the application developer needs deep understanding of: data flows between connectors, compatibility check, multi-threaded data processing, among other mechanisms. The Kurento platform [65] has developed an agnostic component[1] to ease the development of automatic conversion of media formats for dynamic pipeline adaptation. However, this component has specific Kurento's library dependencies for WebRTC media servers that make impractical to use it to develop other type of applications.

[1] `https://github.com/Kurento/kms-core/blob/master/src/gst-plugins/kmsagnosticbin3.h`

6.6 Conclusions

In this paper, we have presented the design, prototype implementation, and evaluation of the Plan, and Execution phases of a proposed context-aware autonomic system to adapt multimedia pipelines. Our evaluation shows that the average time spent to create the variability search space in the Plan phase is in the order of hundreds of milliseconds. The execution of the selected plan is in the order of milliseconds when building the pipeline from scratch, and approximately 1000 times faster when reconfiguring a pipeline and re-utilizing the already instantiated hardware-dependent components. Reconfiguration of pipelines also mitigates the unpredictability of queries in GStreamer compatibility check mechanisms. As future work, we propose to apply the reconfiguration mechanisms to speed up the creation of the variability search space.

Chapter 7

P2 – Autonomic Adaptation of Multimedia Content Adhering to Application Mobility

Authors: Francisco Javier Velázquez-García, Pål Halvorsen, Håkon Kvale Stensland, and Frank Eliassen

Published in: Distributed Applications and Interoperable Systems (DAIS 2018)

Reference in Bibliography: [192]

Abstract: Today, many users of multimedia applications are surrounded by a changing set of multimedia-capable devices. However, users can move their running multimedia applications only to a pre-defined set of devices. Application mobility is the paradigm where users can move their running applications (or parts of) to heterogeneous devices in a seamless manner. In order to continue multimedia processing under the implied context changes in application mobility, applications need to adapt the presentation of multimedia content and their internal configuration. We propose the system DAMPAT that implements an adaptation control loop to adapt multimedia pipelines. Exponential combinatorial growth of possible pipeline configurations is controlled by architectural constraints specified as high-level goals by application developers. Our evaluation shows that the pipeline only needs to be interrupted a few tens of milliseconds to perform the reconfiguration. Thus, production or consumption of multimedia content can

continue across heterogeneous devices and user context changes in a seamless manner.

7.1 Introduction

Multi-device environments with heterogeneous multimedia capabilities are common environments for many people. However, users of multimedia applications can offload their applications or redirect multimedia sessions only to a limited set of pre-defined devices or running environments. This limitation is due to the current paradigm where multimedia applications are designed to start and end execution in the same device. One approach to solve this limitation is to develop applications that adhere to the application mobility paradigm [204]. In this paper, we refer to such applications as *mobile applications*.

Application mobility is the paradigm where users can move parts of their running applications across multiple heterogeneous devices in a seamless manner. This paradigm involves *context* changes of hardware, network resources, user environment, and user preferences. If such context changes occur during an ongoing multimedia session, the application should adapt: (1) the presentation of the multimedia content to fulfill user preferences, and (2) the internal configuration of the application to continue execution in a different running environment.

To move the process of the application from one device to another during runtime, and during an ongoing multimedia session, the needed mechanisms, such as for process migration [140], should be part of DAMPAT. In this paper, we do not address these mechanisms, but focus on the aspects to adapt the presentation of multimedia content.

Multimedia content is composed by a collection of media streams and modalities; e.g., video, audio, and text; which makes a specific *multimedia presentation*. If a mobile application aims to adapt multimedia presentations in a variety of ways, such as bitrate adaptation, modality adaptation, or content retargeting, the more complex it is for developers to design and implement it. Creating complex computing systems that adapt themselves in accordance with high-level guidance from humans (developers or users) has been recognized as a grand challenge, and has been widely studied by the autonomous computing scientific community [100]. Yet, multimedia mobile applications introduce new scenarios and new challenges. For example, in a videoconferencing use case, suppose the user Alice is using a mobile device while commuting. When she arrives in her office, she wishes to continue the same videoconferencing session by moving parts of the application to a dedicated office videoconferencing system. The new challenges in autonomic computing in this scenario are: (1) changes in availability or appropriateness of I/O interfaces to produce or consume multimedia content, (2) changes in application running environment, (3) strict deadlines of multimedia systems, (4) changes in user's physical environment, and (5) changes of user preferences.

It is fair to assume that usability and high QoE are among the main goals of developers of multimedia applications. We translate these goals as a safety predicate based on two requirements: (1) the collection of multimedia streams has to be processed on time and in synchrony to a reference clock, and (2) the configuration of components has to provide a high enough utility to the user, where user utility is defined by a utility function provided by the developer. To satisfy this safety predicate in application mobility, we identify four self-* properties as requirements: (1) **Self-adaptive**: applications should react to changes in the context by changing their safety predicate accordingly. (2) **Self-configuration**: applications should react to context changes, and change the connections or components of the application, to restore or improve the safety predicate. (3) **Self-optimization**: applications should improve (maximize or minimize) the value of a predefined objective function. (4) **Self-awareness**: applications should be able to monitor and analyze its context.

To meet these requirements, we propose the system DAMPAT: Dynamic Adaptation of Multimedia Presentations in Application Mobility. The goal of DAMPAT is two-fold. The first goal is to reduce the development burden when creating context-aware applications that autonomously adapt the presentation of multimedia content. The second goal is to allow users to (easily) influence the selection of the *best* configuration at runtime, where best is defined as the configuration that produces the highest utility according to the current contextual situation and user preferences.

DAMPAT follows the Dynamic Software Product Lines (DSPL) engineering approach [19]. In DSPL, designing a runtime adaptive system is considered to be a variability management problem, where the variability of the system is captured at design time. In our approach, the sequences of components to process multimedia streams are seen as *pipelines*. Therefore, the variability depends on the number of available components, their tuning parameters, and the topology alternatives. This variability creates a combinatorial explosion and makes the problem NP-hard.

The main contribution of this paper is a holistic presentation of the motivation, design, implementation and evaluation of the functional relation between parts of

DAMPAT. This paper presents: (1) the model of available, appropriate and preferred I/O interfaces of users and multimedia-capable devices, (2) how *functional stages* and *functional paths* control exponential growth due to component, parameterization, and topology variability of multimedia pipelines, (3) the definition of high-level multimedia pipelines, and (4) the definition of a multi-dimensional utility function that takes into consideration context changes for decision making of pipeline selection. For completeness, related contributions for DAMPAT in [191] and [193] are also presented.

Results from evaluating a videoconferencing prototype show that the time to create the adapted pipeline from scratch is in the order of tenths of milliseconds in average. The time to reconfigure a pipeline can be as much as 1000 faster than building the pipeline from scratch. Therefore, we conclude that adaptation of multimedia pipelines is a viable approach to seamlessly adapt multimedia content in a variety of ways, e.g., bitrate, modality, and content retargeting (using components such as [161]), in the application mobility paradigm.

In the remainder of the paper, Section 7.2 explains the main challenges of design and implementation decisions of the proposed system. Section 7.3 evaluates the parts of the system that can negatively impact the seamlessness of multimedia mobile applications. Section 7.4 compares DAMPAT with related work. Finally, Section 7.5 concludes the paper.

7.2 The DAMPAT system

Our system adopts the DSPL engineering approach. In order to separate the concerns of DAMPAT, we follow the Monitor, Analyze, Plan, and Execute (MAPE)-K adaptation control loop [113], where *K* is the knowledge created and used across the MAPE phases (see Figure 7.1). Next, we describe in a top-down manner how the Monitor, Analyze, Plan, and Execute (MAPE)-Knowledge (K) loop is applied in Dynamic Adaptation of Multimedia Presentations in Application Mobility (DAMPAT).

7.2.1 Monitor, Analyze, Plan, and Execute (MAPE) phases

Figure 7.1 represents an *autonomic manager*, a *managed element*, *sensors*, and *effectors*. The autonomic manager is a software component configured by human developers using high-level goals. It uses the monitored data from sensors and internal knowledge of the system to plan and execute the low-level actions that are

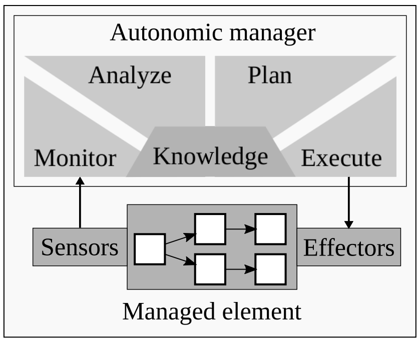


Figure 7.1: Structure of Monitor, Analyze, Plan, and Execute (MAPE)-K control loop

necessary to achieve these goals. The autonomic manager separate the adaptation concerns in four phases: *Monitor*, *Analyze*, *Plan*, and *Execute*, which create and share information (*Knowledge*) that impacts the production or consumption of multimedia content. These phases are explained in Sections 7.2.2 to 7.2.5.

The managed element represents any software or hardware resource that is given autonomic behaviour by coupling it with an autonomic manager. In DAMPAT, the managed element is a multimedia pipeline.

Sensors refer to hardware or software devices that collect information about the running environment of the managed element. DAMPAT also collects information about: the user's available human senses, e.g., a noisy environment prevents a user from producing or consuming audio; user preferences, e.g., always activate close captioning; and modality appropriateness, e.g., no video modality while driving.

The data to asses the availability or appropriateness of modalities can be collected by, for example, setting parameters via a graphical user interface, or complex event processing subsystems. The implementation of these mechanisms, however, is out of scope of this paper. Finally, effectors in Figure 7.1 carry out changes to the managed element.

7.2.2 Phase 1: Monitor

In order for the autonomic manager to relieve humans of the responsibility of directly managing the managed element, the autonomic manager needs to collect data to recognise failure or suboptimal performance of the managed element and effect appropriate changes. Monitoring gives DAMPAT the **self-awareness** property, which it is a prerequisite for **self-optimization** and **self-configuration**. Monitoring involves capturing properties of the environment, either external or internal, e.g., user surroundings or running environment, and physical or virtual, e.g., noise level or available memory. This variation of data sources and data types makes the monitored context *multi-dimensional*.

For the Monitor phase, we group the information that can impact the processing or appropriateness of multimedia presentations in two categories. (1) **User context**: set of I/O capabilities of user to produce or consume multimedia content, physical environment, and user preferences. As for user input capabilities, we consider *hearing*, *sight*, and *touch* senses as interfaces to support audio, video, text, and tactition modalities. As for user output capabilities, we consider *speaking*, and *touching* availabilities as interfaces to support audio, and tactition modalities. User context registers *user preferences*, which are predicates that express additional user constraints or needs. (2) **Application context**: application running environment including I/O capabilities to produce or consume multimedia content. As for device input capabilities, we consider *microphone*, *camera*, *keyboard* and *tangible* (haptic) interfaces. As for device output capabilities, we consider *display*, *loudspeaker*, and *tangible* interfaces. The model also contains software and hardware descriptors for dependencies of pipeline components. Software descriptors include the available software components to build multimedia pipelines, such as encoders, parsers, and encryptors. Hardware descriptors include CPU, GPU, battery, memory, and network adapters.

The design of DAMPAT also takes into account context that impacts the appropriateness of modalities in a given situation, namely, *current activity*, *geographical location*, *physical environment*, *date*, and *time*. The information needed to estimate the modality appropriateness is taken from both, user- and application-context. The monitored data is part of the knowledge K in DAMPAT.

7.2.3 Phase 2: Analysis

We say that an application is in a legal or consistent configuration in a given context, when the corresponding safety predicate holds. A safety predicate in application mobility is not only violated by bugs or failures in software or hardware, as in traditional scenarios in autonomic computing, but also by changes in user and application context, that change the initial high-level goal of the application. For example, when a user changes preferences from audio to text modality due to a noisy environment or when an audio card is not longer available in a multimedia session after an application has moved.

To meet the **self-adaptive** requirement in DAMPAT, we declare two characteristics of safety predicates: (1) safety predicates hold if a pipeline configuration is adequate for the available resources of the application running environment so that buffers arrive on time in the final sink, and (2) safety predicates might change with changes in context.

Therefore, if the user changes her environment or preferences, the autonomic manager treats such changes as a threat to the safety predicate and addresses them. In a more obvious manner, if the application moves to another device where the initial configuration cannot continue execution, the autonomic manager addresses this problem as well. The **self-optimization** requirement is met by objective functions implemented in components. For example, a DASH (Dynamic Adaptive Streaming over HTTP) component that proactively checks the available resources to optimize its parameterization and process the highest bitrate.

The problem-diagnosis component in the Analysis phase analyzes the data collected in the Monitor phase. This component can evaluate whether the safety predicate holds. If the safety predicate is violated, it means that a problem is detected, and the Plan phase is started. The implementation of the problem-diagnosis component can be, for example, implemented based on a Bayesian network. This implementation is left as future work.

The current design of the Analysis phase of DAMPAT, takes into consideration the monitored data of the device where an application starts execution (source), and the device where the application will be moved to (destination). As future work, we plan to incorporate the special purpose negotiation protocol in [10] to aggregate the monitored data of all the surrounding devices to which an application can move.

7.2.4 Phase 3: Plan

In the Plan phase, the autonomic manager creates variants of multimedia pipelines, and selects the best one among the ones that guarantee to hold the safety predicate in the current context. The Plan phase addresses the challenge of combinatorial explosion of pipeline variants caused by compositional and parameterization variability. In the current state of DAMPAT, the Plan phase assumes infinite resources of application running environment, and do not consider other applications running in the same device.

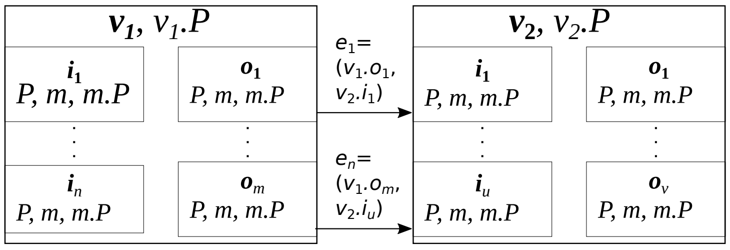


Figure 7.2: Multigraph that shows vertices $v_1, v_2 \in V$ representing pipeline components. P represent a set of properties, i and o represent input and output connectors, $m \in M$ the supported modalities by the connectors, and $e \in E$ represent links or pipes between connectors.

Multimedia pipeline model

Multimedia pipelines are built with components that are linked with compatible connectors, and process streams in a sequential order. Multimedia pipelines can be modeled as directed acyclic multigraphs $G = (V,E)$. In this abstraction, V is the set of vertices v that represent the pipeline components, and E is the set of edges e that represents the connection or pipe between the output and input connectors of two vertices. Each edge has a modality type m, and multiple edges $(e \in E)$ connecting components of the same components can have different modalities. Therefore, multigraphs have a set of modalities M.

Figure 7.2 illustrates a simplified version of two connected pipeline components representing a multigraph G. Each component v has a set of input connectors $v.I$ and output connectors $v.O$. *Connectors* are the interfaces of components. Data flows from one component's output connector $v.O$ to another component's input connector $v.I$. The specific data type (modality) that the component can handle is described in the component's connectors.

Pipeline components for the same functionality might have different implementations, for example; (1) the components `vp8dec` and `avdec_vp8` are two different implementations of the VP8 decoder, and (2) the components `glimagesink`, and `waylandsink` are two different implementations that differ in hardware offloading and memory allocation (among many other differences). Therefore, in the multimedia pipeline model represented in Figure 7.2, each component v can have more than one implementation candidate, and some components can dynamically (on-demand) create a set of input ($i \in v.I$) or output ($o \in v.O$) connectors. We refer to this configuration variability as *compositional variability*. In a similar manner, every component has *parameterization variability* due to assignable

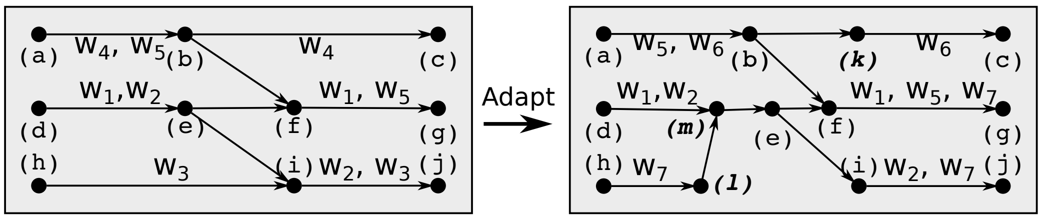


Figure 7.3: Graph abstraction of multimedia pipeline of one videoconferencing peer before and after adaptation. On the left, pipeline that consumes and produces video and audio. On the right, pipeline that consumes video and text from a peer that cannot process audio, this pipeline allows its user to produce and consume audio by changing the text-to-audio and audio-to-text modalities. The vertices represent the following components: (a) `networksrc`, (b) `demuxer`, (c) `audiosink`, (d) `webcamsrc`, (e) `splitter`, (f) `videomixer`, (g) `videosink`, (h) `audiosrc`, (i) `muxer`, (j) `networksink`, (k) `text-to-audio`, (l) `audio-to-text`, and (m) `text-overlay`. $\{w\}_1^7$ represent functional paths.

property values of components ($v.P$), connectors ($i.P$ and $o.P$) and modalities ($m.P$). Compositional and parameterization variability can create a rapid growth of complexity due to combinatorial explosion.

Typically, multimedia presentations are composed by more than one multimedia stream, e.g., video and audio stream. In our multimedia model, a *path* is a sequence of successive edges through the graph (where a vertex is never visited more than once) for a given stream. In complex multimedia pipelines, a stream can be split or mixed, increasing or reducing the numbers of streams. For example, a video stream that is split to be (1) rendered in a display, and (2) sent over a network card, or a video and audio streams that are multiplexed to be sent through a network card. Therefore, we define the term *functional path* as the path w of one stream from its original source to its final sink. For example, in the left pipeline of Figure 7.3, there are five functional paths, w_1, w_2, w_3, w_4, and w_5, where paths w_4, w_5 share source (a), w_1, w_2 share the source (d), w_1, w_5 share sink (g), and w_2, w_3 share sink (j). The right part of the figure is explained in Section 8.3.

Pipelines have a set of behavioral and interaction rules that aim to minimize the processing latency of the stream in the pipeline. Mechanisms to create, manage and dynamically reconfigure multimedia pipelines include: connector compatibility check, connector linking, stream flow control to handle delayed buffers in sinks due to limitations in local resources or bandwidth, pipeline state management, components instantiation, and memory allocation type check to avoid memory copying. To the best of our knowledge, GStreamer [86] is the only free

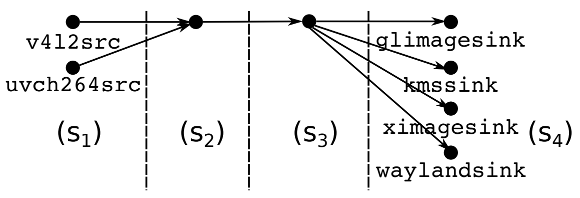


Figure 7.4: Example of functional path (w) that captures video from a webcam and renders it in a display. In this example, w has four functional stages ($\{s\}_1^4$) and eight possible path combinations.

and open source, multi-platform, multimedia framework actively implementing and maintaining these mechanisms. Therefore, we leverage GStreamer pipelines in DAMPAT.

Control of combinatorial growth due to compositional and parameterization variability

We arrange functional paths (W) in a sequence of *functional stages* ($s \in S$) that group components by functionality, e.g., file sources, demuxers or decoders. Functional stages act as architectural constraints to enforce directed graphs, and they avoid unnecessary checks of connector's compatibility, which are most likely to fail. An architectural constraint is defined as the design knowledge introduced by the application developer with the purpose to reduce combinatorial growth (by limiting configuration variability).

For example, in Figure 7.4, the developer defines a functional path (w) to capture video taken from a webcam, and render it in a display. This functional path is defined with four functional stages (s_1, s_2, s_3, s_4). The functional stage s_1 groups the components that capture video, stage s_2 is a specific component to fix the desired output of s_1, s_3 does conversion of color space, and s_4 groups the components to render video. In this example, since there are two candidates in s_1, and four candidates in s_4, there are 8 possible functional paths.

Functional stages are defined at different *levels*, where deeper levels filter components more accurately. In this way, application developers can define high-level architectures of multimedia pipelines without knowing the details of each functional stage. For example, developers can define a *pre-processing* stage that automatically includes components of the type of protocol handlers, parsers, and video converters. For further details about this approach, the reader is referred to [191].

GStreamer multimedia components and enumerated parameters have a rank to describe their priority with competing candidates. Functional stages is a list of stages, where each stage is a list of candidates sorted by rank, just as the vanilla auto plugin strategies in the GStreamer do, and build the variability search space of functional paths by sequentially testing each sorted candidate. As a result, the produced search space is a sorted list of functional paths.

Linking the connectors across the defined functional stages produces a unix-style configuration file that is part of the knowledge of DAMPAT. This file contains the settings of all configuration options for every component in the functional stage. Listings 7.1 and 7.2 show snippets of the configuration file for one functional path in Figure 7.4.

Listing 7.1: Snippet 1 of `w1.conf`

```
[functional-path]
name=webcam2display
vertices=videosrc,filter,\
tee,queue,glimagesink
```

Listing 7.2: Snippet 2 of `w1.conf`

```
[vertex videosrc]
name=v4lsrc0
output-conn=v4lsrc0.src.0
device=/dev/video0
```

Control of functional path combinations

Due to the compositional variability in functional stages, functional paths may have a set of alternative paths, consequently, alternative topologies. In order to restrict path combinations, the application developer can introduce an architectural constraint with specifying the bound of allowed path combinations per functional path. The combinatorial growth of this approach is evaluated in Section 7.3.1.

To enforce the path combination constraint, the autonomic manager computes the Binary Reflected Gray Code (BRGC) algorithm. The output of the BRGC algorithm is a set of subgraphs $G' = \{g\}_1^n$ that creates the variant search space. Each element $g \in G'$ represents a pipeline that can be configured in the Execute phase. Each pipeline (g) has the set of properties ($P \in v$) of each component ($v \in g$), the set of modality types ($M \in g$) processed by the pipeline, the properties of each modality ($P \in M$), and the set of edges $E \in g$. In practice, the description of each g is stored in a configuration file similar to Listings 7.1 and 7.2, but its values are the location of files describing the set of functional paths ($W \in g$). G' is part of the knowledge base of DAMPAT, and its elements are used as input for the utility function used in the decision making process.

Variant selection

The autonomic manager evaluates the variants in the search space and selects the alternative that matches best the goals defined by the application developer, user preferences, and contextual information. The challenge in this selection is how to define high-level goals and how to trade off conflicting contextual information. High-level goals are usually expressed using event-condition-action (ECA) policies, goal policies or utility function policies [100]. ECA policies suffer from the problem that all states are classified as either desirable or undesirable. Thus, when a desirable state cannot be reached, the system does not know which among the undesirable states is least bad. Goal policies require planning on the part of autonomic manager and are thus more resource-intensive than ECA policies. Utility functions allow a quantitative level of desirability to each context. Therefore, we use multi-dimensional utility functions.

The proposed multi-dimensional utility function [191] is composed of functions defined for the properties that describes the pipeline ($g.P$). Developers of pipeline components define and implement the component and its utility function. Since the overall pipeline utility is calculated based on the components that form the pipeline, the more utility functions are implemented in the components, the better overall estimation can be calculated. Utility functions take as argument two property-value tuples, one argument represents the user preference ($u.p$), and the other argument is the property value ($g.p$) obtained from the running environment, e.g., hardware characteristics or metadata of stream. As a result, the signature of utility functions in components are of the form $ut(u.p, g.p)$.

If a modality is unavailable or inappropriate for a user in a given context, the modality is marked as negative. Therefore, pipeline variants matching negative modalities do not provide the highest utility, and thus they are not selected. One analogy to see this approach, is to think of the human senses as connectors (interfaces). In this analogy, DAMPAT matches the best compatibility between the possible pipeline configurations to use the computer's interfaces, and the human's interfaces. Figure 7.5 illustrates this analogy in a oversimplified pipeline that processes video and audio modalities.

Weights (we) Weights are provided by users to (easily) influence the selection of the configuration at runtime. Weights help to trade off conflicting interests, and they can be seen as ranks or importance associated to a property, i.e., $u.p.we$. For example, suppose a user prefers `video-resolution=4K` (2160 progressive)

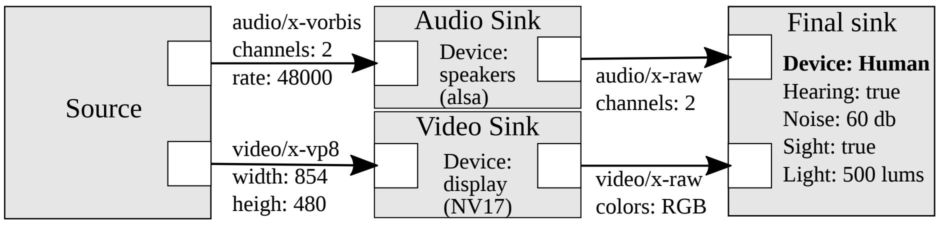


Figure 7.5: Oversimplified pipeline to make an analogy of a human consumer as a component in a multimedia pipeline. In this analogy, the input connectors (interfaces) of a human consumer are the hearing and sight senses. DAMPAT selects the pipeline variant with connectors that are compatible with the available and appropriate user interfaces in a given context.

and `framerate=60fps`. In case a device can reproduce either 1080p at 60 fps, or 4k at 30 fps, weights are used to rate the alternatives. Thus, the resulting weighted multi-dimensional utility function is $\Upsilon(u, g) = \sum_{j=1}^{l} ut(u.p_j, g.p_j) \cdot u.p_j.we$ [191].

Finally, if all the pipelines in the variability search space provide 0 utility, DAMPAT interprets this situation as if adaptation is impossible for the given context. If the application cannot continue execution in the current running environment, DAMPAT stops the application.

7.2.5 Phase 4: Execute

The task of the this phase is to safely introduce, remove, or re-configure components in the pipeline according to the selected subgraph g, i.e., pipeline variant with highest utility for a given context. g contains the description of the pipeline variant to be executed (described in Section 7.2.4). Then, the autonomic manager decides between create the pipeline from scratch or reconfigure it. The Execute phase meets the **self-configuration** requirement in DAMPAT.

The autonomic manager compares the current pipeline configuration (if already instantiated) with the new selected variant. In our implementation design, the autonomic manager executes the `diff` Linux command with the `.conf` files from the current and new graph descriptors as arguments. If the output of `diff` includes changes in source components in the pipeline, the new variant is instantiated from scratch, because new sources typically require several changes that are more complex to automate, and thus are prone to errors.

Dynamic reconfiguration

If a component is removed while it is processing a buffer, the thread processing the stream can potentially enter in a deadlock state, because some other component(s) in the path might indefinitely wait for the expected data to arrive. To prevent this situation, the autonomic manager blocks the data flow in the preceding connector of the component that will be removed, and installs a callback to be notified about the state changes in the data flow. After changing components, the state of all components is synchronized to avoid deadlocks.

A potential race condition when reconfiguring pipelines occurs when a component in the pipeline waits for some timestamp or other specific data that was in the buffer or a just removed component. The adaptation manager handles this situation by flushing the buffers of the components to be removed. If the Execution phase fails to instantiate the selected variant, DAMPAT blacklist the just failed variant, and runs the variant selection process again.

State preservation for stream processing is achieved by reading the timestamps of the stream. We assume that states of components and pipelines are preserved when moving between devices. This can be achieved, for example, by implementing component's interfaces that retrieve and store the state of the components.

7.3 Evaluation

In this section, we present and discuss the evaluation of the time overhead that has a direct impact in multimedia session interruption. In principle, this overhead is the time to select and execute the plan, either by instantiating a pipeline from scratch or reconfiguring it. However, if the variability search space is not ready by the time adaptation is needed, its creation can also add interruption time. Results of experiments are from two evaluations from our previous work in [191], and [193].

For completeness, we briefly describe both prototypes and the experiments. In evaluation 1 [191], we evaluate the Plan phase to adapt a video player prototype application that consumes video and audio modalities. The experiments evaluate the creation of the search space with four and six functional stages, and an initial repository of 1379 pipeline components.

In evaluation 2 [193], we evaluate the Plan and Execution phases of a videoconferencing prototype application that simulated the production and consumption of: video, audio, and text modalities. The pipeline in this evaluation is of a peer videoconferencing application that has to adapt since (for any reason) its peer cannot process audio any longer. However, the user of this pipeline prefers to interact with the audio I/O interfaces of the device. The initial and reconfigured pipeline of this evaluation is the same as in Figure 7.3. The initial repository is of 1420 pipeline components.

As a testbed, evaluations 1 and 2 use the same computer that resembles hardware characteristics of commodity hardware. The computer is a MacBook Pro 7,1 with Intel Core 2 Duo CPU P8800 at 2.66GHz running the 64-bit Ubuntu 17.10 operating system.

7.3.1 Plan phase

In this section, we discuss results from our previous work ([191] and [193]) to create the variability search space, and to select the variant with highest utility. The main scaling factors that influence the time spent when creating the search space are: (1) the time to instantiate components with hardware dependencies, (2) the query handlers in GStreamer components to check the processing capabilities of connectors, (3) the length of the pipeline, (4) the number of functional stages per functional path, and (5) the number of candidates per stage.

Results from the evaluations show that the time to create the entire variability search space is between the order of a few seconds and hundreds of milliseconds. Observations about the number of queries are: number of queries does not have a linear correlation with the number of functional stages or number of components in each stage due to the different implementations of query handlers in the involved components, and number of queries increases as the path length increases due to the recursion of queries.

To evaluate the scalability issues when combining functional paths, we use binomial coefficients to calculate how many unsorted combinations exist to select $k \geq 0$ path configurations. That is $\binom{n}{k} + ... + \binom{n}{0}$, where n is the cardinality of the set of configurations for a specific path definition. As a result, when the developer decides to restrict functional path configurations to one ($k = 1$) in an application with three needed paths; e.g., video rendering, video transmission, and audio transmission ($n = 3$), the combinatorial growth is reduced to the polynomial form of $\mathcal{O}(n^k)$, i.e., $\mathcal{O}(3)$.

Evaluation and analysis of the multi-dimensional utility function, described in Section 7.2.4, shows that its complexity is linear. Since the maximum number of pipeline variants in our experiments are below 300, a brute force approach to find the variant with the highest utility does not introduce intolerable service interruption. However, greedy techniques, such as Serene Greedy [168], should be implemented in DAMPAT to tackle larger search spaces. The implementation of greedy techniques, however, is left as future work.

7.3.2 Execution phase

In this section, we discuss results from our previous work ([191] and [193]) that evaluates the time to execute a plan by two means: by instantiating a pipeline from scratch or by reconfiguring it. The main factors when instantiating a pipeline from scratch are the same as in the Plan phase, but not when reconfiguring a pipeline. Reconfiguration of pipelines is faster mainly due to the re-utilization of already instantiated components with hardware dependencies, and the need for less queries to check compatibility of components' connectors. However, the reduction of queries does not correlates linearly. The removal of functional paths reduces the number of queries drastically, in some cases 0 queries needed, as opposed to instantiating the adapted pipeline from scratch. Therefore, further implementation of DAMPAT should aim at removing functional paths only by reconfiguration.

Results show that the execution of a plan (involving functional paths with similar characteristics as in Figure 7.3) is under 10 ms when instantiating a pipeline from scratch. There is a clear pattern of approximately 1000 times faster (from tens of milliseconds to tens of microseconds) when reconfiguring a pipeline, if the already instantiated hardware-dependent components are reused.

The speed gain from pipeline reconfiguration over instantiating pipelines from scratch is applicable when adaptation occurs in the same device. Clearly, if an application is moved from one device to another, the components with hardware dependencies have to be initialized in the destination device. Therefore, in such mobility cases, there are no advantages in reconfiguring a pipeline.

Reconfiguration in the same device is, however, still a valid use case in peer to peer mobile applications, such as in the videoconferencing use case illustrated in Figure 7.3. Pipeline reconfiguration can be also very advantageous when creating the variability search space, specially in the current design of DAMPAT where the variability search space is created based on local components only. In order for DAMPAT to know whether reconfiguration is a better alternative (than instantiation from scratch), pipeline components must be annotated to indicate whether they have hardware dependencies or not. This annotation and the creation of the variability search space using the reconfiguration mechanisms are future work.

7.4 Related Work

MUSIC [93] is a development framework for self-adapting applications in ubiquitous computing environments; it follows the MAPE-K reference model, and it uses utility functions for adaptation decision making. MUSIC combines component-based software engineering with service-oriented architectures (SOA) to allow applications on mobile devices to adapt to and benefit from discoverable services in their proximity. Applications in MUSIC can offload services to devices in close vicinity; these close devices must, however, have pre-installed the MUSIC middleware and application-specific components. Therefore, the application developer has to be aware of the characteristics of the devices where applications can move. As a result, the set of devices constituting the ubiquitous environment is defined at design time of the application. Hallsteinsen et al. [93] recognized that support for multimedia content adaptation in a challenging research alley, and left it as future work.

PLASMA [122] is a component-based framework for building adaptive multimedia applications. This framework relies on a hierarchical composition, similar concept to levels in functional stages (described in Section 7.2.4), and a reconfiguration model, similar to the Execute phase (Section 7.2.5). The authors describe at a high-level the mechanisms needed to build and reconfigure pipelines. However, they do not discuss the needed mechanisms to process multiple media types in synchrony. Therefore, we regard their design valid for adaptation of only one stream. PLASMA does not handle any scalability issue due to parameterization or compositional variability. PLASMA is implemented in DirectShow (moved to Windows SDK in 2005), which implies support for devices running Windows operating systems only. Adaptation policies in PLASMA are based on event-condition-actions (ECA), and they are triggered on changes of hardware resources only, e.g., bandwidth fluctuations, but not changes between devices, therefore PLASMA-applications do not adhere to the application mobility paradigm.

Infopipes [27] provides abstractions to build distributed streaming applications, that adapt based on resource monitoring, such as CPU and bandwidth. Therefore, adaptation is achieved by adjusting the parameters of components only, and it limits the adaptation types that can be achieved with compositional variability. The authors define pipelines with pipes, filters, buffers, and pumps, but do not define the mechanisms to process multiple streams in synchrony.

7.5 Conclusions

We have identified the self-adaptive, self-optimization, self-configuration, and self-awareness properties as requirements for multimedia applications to adapt the presentation of multimedia content across the multimedia-capable devices that surround users. To ease the development of multimedia applications that meet these requirements, we have presented DAMPAT, which follows the MAPE adaptation control loop, and DSPL engineering approach. DAMPAT enables application developers and users to describe the application goals at their level of expertise via: configuration files (functional stages, and functional paths), user preferences, and importance of preferences. This approach allows users of mobile applications to take advantage of heterogeneous devices that were unknown at design time. DAMPAT makes decisions at runtime on how to adapt multimedia presentations; it enables modality adaptation, and any other adaptation technique implemented in the pipeline components such as bitrate adaptation, or content retargeting.

The main contribution of this paper is the holistic presentation of the motivation, design, implementation, and evaluation of DAMPAT. Evaluation shows that the average time spent to adapt multimedia pipelines is in the order of milliseconds. This delay is acceptable when users of mobile applications have to physically move their attention and control from one device to another.

As future work, we plan to explore the creation of a model to quantify the effects of the previous configuration when reconfiguring a pipeline; as first approach, we suggest to do analysis of variance, and regression in experiments to process more than three media types. To create this model, we plan to investigate what are the currently available GStreamer components that can be instantiated in a sample of multimedia devices in typical homes, offices and public transportation in industrialized countries. Additionally, we plan to add more managed elements to adapt different parts of mobile applications, e.g., reconfiguration of endpoint connections.

Chapter 8

P3 – DAMPAT: Dynamic Adaptation of Multimedia Presentations in Application Mobility

Authors: Francisco Javier Velázquez-García and Frank Eliassen

Published in: 2017 IEEE International Symposium on Multimedia (ISM)

Reference in Bibliography: [191]

Abstract: Application mobility is the mobility type where users can move their running applications across multiple heterogeneous devices in a seamless manner. This mobility involves dynamic context changes of hardware, network resources, user environment and user preferences. State-of-the-art adaptive applications adapt in multiple ways to a subset of scenarios in application mobility, however, adaptation of multimedia presentations is rather limited to the selection of pre-processed variants of multimedia presentations. We propose DAMPAT for GStreamer to autonomously adapt multimedia pipelines for a given context. DAMPAT adopts the Dynamic Software Product Line (DSPL) engineering approach, and separates the concerns of the system by following the Monitor, Analyze, Plan, and Execute (MAPE) model. DAMPAT limits the combinatorial explosion of pipelines variability by introducing architectural constraints. Results from evaluation show that DAMPAT can adapt multimedia pipelines adhering to the application mobility paradigm in the order of hundreds of milliseconds.

8.1 Introduction

Application mobility [204] impacts the means to produce or consume multimedia content when an application is moved into different running environments. On the one hand, different device capabilities might have different I/O interfaces, and on the other hand, user context determine the appropriateness of certain media presentation modalities. In such situations, multimedia content, composed of a collection of media streams, i.e., a *multimedia presentation*, needs to be adapted.

Application mobility is very advantageous for users and developers of multimedia applications. Users can take advantage of different multimedia device capabilities as they become available in their environment, without breaking ongoing multimedia sessions. Developers can create applications that are able to execute in, or move to, devices with characteristics that were unknown at design time.

One problem to address when developing multimedia applications that adhere to application mobility is how to adapt multimedia presentations. Multimedia presentations can be adapted by different adaptation types, namely *fidelity*, *modality*, *content-based*, or *retargeting* adaptation. However, applying only one adaptation type is not enough to adapt to the very large diversity of devices and situations while satisfying user preferences and QoS requirements. Yet, the more adaptation types are applied to a multimedia presentation, the more variants of it can be created, and this situation can rapidly become a scaling and management issue.

Multimedia presentations are processed by sequentially connected components, a.k.a. *multimedia pipelines*. Depending on the needed adaptation type, streams in multimedia presentations can be adapted either by tuning the parameters of the components in the pipeline, changing the components themselves, or changing the topology of the pipeline. The mechanisms needed to manage multimedia pipelines have been addressed by different multimedia frameworks. However, to the best of our knowledge, GStreamer [86] is the only open source framework that allows the development of pipeline components and full-fledged multimedia applications. GStreamer includes an implementation for pipeline generation on player startup, but this cannot be used out-of-the-box in applications adhering to application mobility because; 1) pipelines are generated at startup with modality selection as only reconfiguration alternative during runtime, and 2) the pipeline generation is designed to consume multimedia content, not to transcode or produce it.

The problem statement we address in this paper is: how to automate the adaptation of GStreamer multimedia pipelines in application mobility, while limiting the combinatorial growth due to 1) the ever growing number of pipeline components, and 2) the large number of tunable parameters of pipeline components. In addition, pipeline selection by comparison among a very large number of alternatives can produce an unacceptable time overhead in *everyday devices*, which can deteriorate the quality of experience of users in multimedia applications. Therefore, pipeline selection should be done in a timely manner.

We propose the runtime adaptive system called DAMPAT: Dynamic Adaptation of Multimedia Presentations in Application Mobility for GStreamer. The goal of DAMPAT is two-fold. First, it reduces the development burden when creating autonomic context-aware adaptive pipelines that adhere to the application mobility paradigm. Second, users can easily influence the selection of the pipeline variant that produces the highest utility for the user according to the current contextual situation and user preferences.

Our contribution in this paper is the design, prototype implementation and evaluation of that part of the system creating multimedia pipeline variants, and selecting the best one for a given context. Results from performance measurements combined with performance analysis of the involved algorithms show that adaptation of multimedia pipelines is a viable approach for adaption of multimedia presentations in the application mobility paradigm.

8.2 Design and implementation

Application mobility goes hand in hand with context changes. In order to adapt multimedia streaming presentations to these changes, we design the runtime adaptive system DAMPAT, which includes a context-aware *autonomic adaptation manager*, and a *managed multimedia pipeline*. By *context-aware*, we refer to the extensive and continuous use of any information that characterizes the user surroundings or application running environment, which impacts the processing of multimedia presentations.

DAMPAT follows the DSPL engineering approach [19]. In DSPL, designing a runtime adaptive system is considered to be a variability management problem, where variability of the system is captured at design time, and the *best* product variant is selected at runtime. We define *best* as the variant that produces the highest utility according to the current contextual situation. In order to break down the concerns for building the adaptation manager, we follow the *MAPE* adaptation control loop [113].

In this paper, we concentrate on the *Plan* phase of the MAPE model. In this phase, the adaptation manager creates variants of *valid* multimedia pipelines and selects the best one for a given context. By valid multimedia pipelines we mean a pipeline with adequate configuration for the available resources, so that buffers arrive on time at the final sink.

8.2.1 Multimedia pipeline model

Multimedia pipelines are built with components that process streams in a sequential order. Mechanisms to create, manage and dynamically reconfigure multimedia pipelines include: connectors compatibility check, connectors linking, stream flow control to handle delayed buffers in sinks due to limitations in local resources or bandwidth, pipeline state management, components instantiation, and memory allocation type check to avoid memory copying. To the best of our knowledge, GStreamer is the only open source multimedia framework actively maintaining these mechanisms. Therefore, we implement DAMPAT for GStreamer pipelines.

Multimedia pipelines can be modeled as directed acyclic multigraphs $G = (V,E)$. In this abstraction, V is the set of vertices that represents the components in the pipeline, and E is the set of edges that represents a connection or pipe between the output and input connectors of two pipeline components. Each edge has a modality type m, and multiple edges can have different modalities. Therefore, multigraphs have a set of modalities M.

Pipeline components might have different implementations, e.g., components with and without hardware offloading, or components with and without use of homogeneous shared memory allocation. Therefore, each component can have more than one candidate, which is referred to as *compositional variability*. In a similar way, every vertex has *parameterization variability* due to assignable property values of vertices, connectors and modalities. Compositional and parameterization variability can create a rapid growth of complexity due to combinatorial explosion. To reduce the variability growth, we limit the possible configurations by allowing the designer of the multimedia application to introduce architectural design knowledge, which is commonly referred to as *architectural constraints* [77].

The use of architectural constraints reduces the number of pipeline variants to consider when searching for the best variant. For the variant selection, we apply multi-dimensional utility functions that allow DAMPAT to take into consideration

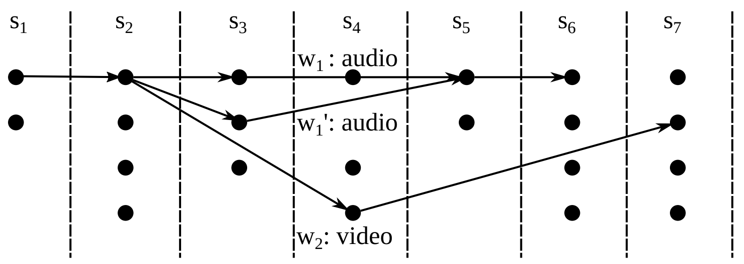


Figure 8.1: Functional stage $\{s\}_i$, and paths w_1, w_1' and w_2

Table 8.1: Levels of functional stages

L. 1	pre-processing						retargeter				post processing					
L. 2	source handler		Input Format Handler			converter	adaptation type			Filters		Format handler		Sink Handler		
L. 3	protocol handler	source handler	parser	demuxer	decoder	video converter	modality adaptation	content adaptation	fidelity adaptation	stream selector	mixer	encoder	muxer	payload encoder	session manager	sink handler

multiple user preferences, and resolve conflicting or mutually dependent concerns among user preferences, user context, and application running environment.

8.2.2 Control of combinatorial growth due to compositional and parameterization variability

We arrange the multigraph abstraction in a sequence of *functional stages*, that group components by functionality, e.g., file sources, demuxers or decoders. Functional stages act as architectural constraints to enforce directed graphs, and they avoid unnecessary checks of connectors compatibility, which are most likely to fail, e.g., compatibility checks between components in stages s_1 and s_7 in Figure 8.1. Combinatorial growth is further limited by restricting how many distinct *paths* of sequentially connected components can process one stream, e.g., only one path is allowed to process audio. Control of path combinations is discussed in Section 8.2.3.

Functional stages are defined at different levels, where deeper levels filter components more accurately. For example, Table 8.1 represents stages at three levels for a video server pipeline. In the table, stage *pre-processing* lists components that match with metadata descriptors such as: *protocol handler, parser, demuxer* and *decoder*. Then, if the neighboring output and input connectors are compatible, they are linked.

If the adaptation manager creates one pipeline variant for each combination of possible values of the properties of components, connectors, and modalities, the variant search space is in principle in the order of a million different pipelines. Therefore, in order to limit the parameterization variability, DAMPAT allows the application developer to specify a list of typical values for a set of common scenarios.

8.2.3 Control path combinations

Due to the compositional variability in functional stages, multimedia streams may have a set of alternative paths $W = \{w\}_i$. Alternative paths for the same modality are identified with the prime symbol, as in $'_1$ in Figure 8.1. However, a pipeline with such a topology, e.g., decoding one stream to multiple different formats in parallel, is unlikely in typical multimedia applications for the application domain we target.

In order to remove pipelines with undesirable path combinations, DAMPAT introduces an architectural constraint to limit the path combinations, where the upper bound of allowed path combinations is specified by developers of applications. We use binomial coefficients to calculate how many unsorted combinations exist to select $k \geq 0$ paths for the same modality. That is, $\binom{n}{k} + ... + \binom{n}{0}$, where n is the cardinality of the set of paths Γ for the same modality m. As a result, when the developer defines k, the combinatorial growth is reduced to the polynomial form of $\mathcal{O}(n^k)$.

Equation 8.1 generalizes the architectural constraint to limit path combinations by denoting the subset of paths W' for the same modality, such that the cardinality of W' is less or equal to the number of allowed paths k specified by the developer. Then, the number of valid pipelines in the variant search space, i.e., $|G'|$, is the product of the allowed combinations of paths per modality. In case that a developer defines as valid path combinations to only those with at most one path per modality ($k = 1$), the creation of the variant space has multi-linear growth.

$$|G'| = \prod_{m \in M} (|\{W' \subseteq \Gamma_m \mid |W'| \leq k\}|) \tag{8.1}$$

To enforce the path combination constraint, the adaptation manager generates the power set $\mathcal{P}(W)$ in the form of bit strings of all paths in graph G by using the Binary Reflected Gray Code (BRGC) algorithm [126, p. 174], and then counts how many paths with the same modality exist in a subset. If a modality counter is greater than k, the subset is invalid. BRGC has exponential growth $\mathcal{O}(2^{|W|})$, but the resulting search space is constrained to the complexity resulting from Equation 8.1.

The resulting set of subgraphs G' creates the variant search space. Each element subgraph $\in G'$ contains the description of the properties of each pipeline component in subgraph, the set of modalities occurring in subgraph, the properties of each modality, and the set of edges E in G'. The elements of G' are used as input for the utility function used in the decision making process.

8.2.4 Variant selection

The adaptation manager has to select one variant from the variant search space. The challenge in this selection is to trade off conflicting user preferences and contextual information. For this purpose, we adopt multi-dimensional utility functions from [2].

In DAMPAT, a multi-dimensional utility function is composed of functions defined for each component property. Utility functions are defined by the developer of the component, and they take two arguments; the preferred property value $u.p$ specified by the user u, and the corresponding property value $g.p$ provided by the component in the pipeline variant being evaluated. The signature of a dimensional utility function is of the form $ut(u.p,g.p)$.

Users might not provide a preferred value for every single property, therefore, the adaptation manager sets a `any` value for such cases. In this way, developers of utility functions can describe which property values give a higher utility even when there is no user preference. For example, the utility function $ut(any, channels = 6)$ could be defined to give a higher utility than $ut(any, channels = 2)$ in an audio sink component. Utility functions return floating point values between 0 and 1 when evaluated.

Finally, the adaptation manager computes the overall utility of the pipeline variant as the weighted sum of the dimensional utility functions, where each weight $u.p.we$ indicates the importance of each preferred property $u.p$. By default all properties are equally important, but a user may change this by adjusting the weights. DAMPAT ensures the sum of the weights is always equal to one. When a user assigns a preferred value to a property, the adaptation manager assigns the default weight as follows.

Every already assigned weights are divided by the number of weights plus 1 (for the new preference). The quotient is subtracted from its dividend, and the result becomes the new value of the previously assigned weight. The sum of the quotients of all previously assigned weights is the value for the new generated weight value. In this way, the manager preserves the relation to previous adjustments of importance, and keeps the sum of weights to 1.

The resulting utility Υ from all involved utility functions is given by Equation 8.2, where $p_j, j = 1, \ldots, l$, are the properties. Equation 8.2 has linear complexity.

$$\Upsilon(u, g) = \sum_{j=1}^{l} ut(u.p_j, g.p_j) \cdot u.p_j.we \tag{8.2}$$

8.2.5 Linking connectors

The prototype leverages GStreamer `1.13.0 (GIT)`. The adaptation manager links components of neighboring functional stages if the intersection of modality types and modality properties of output and input connectors is not empty. Compatibility checking in GStreamer components is done by intersecting the modalities and their properties of connectors. Therefore, the larger the number of supported modalities and properties, the more operations to obtain the intersection. The current approach in GStreamer to avoid a costly check of compatibility is to group values in ranges when possible, e.g., `framerate : [0, 2147483647]`. Consequently, when a linked pipeline is requested to process a stream, GStreamer must instantiate every component sequentially to double check whether the specific requested parameterization is supported.

In order to check compatibility between connectors, the adaptation manager uses the GStreamer query `query-caps` to check the processing capabilities of connectors. This query is recursive in the way that a received query in a component is sent to the next component in the pipeline to make sure the next connectors can be configured in a compatible manner. Then, if compatible, a query is sent to the next component down the pipeline, and so on.

A second GStreamer query called `accept-caps` is used to confirm whether the format of the stream can be handled by the component. If the developer of the component does not implements a proper handler for `accept-caps`, the default behavior of GStreamer is to create a recursive `query-caps` query again. Therefore, the current approach of GStreamer to register parameterization variability in components, can potentially introduce scalability issues in the autonomous creation of the variability search space. The performance effect of this limitation is evaluated in Section 8.3.

In principle, compatibility checking can be done by checking the properties of GStreamer elements in their metadata registry. However, some elements such as convertors, decoders and encoders define their output connector based on the input stream *and* required output, because not all input streams may have the necessary metadata to help determine the output format. Therefore, either manual configuration or a specific architectural constraint, such as a filter component with well defined properties, is needed for the some pipelines.

A consequence of this GStreamer limitation is that the adaptation manager cannot create the variability search space before knowing the input stream, and the manager might not be able to autonomously select components that require manual configuration. One way the manager can mitigate this problem is by; 1) sending the metadata of the input stream (as soon as the manager knows it) to the accessible devices where the application can potentially be moved, and 2) restricting the output connectors based on the available final sink components in the device. In this way, the creation of the search space can be done before adaptation is needed.

8.3 Evaluation

We evaluate DAMPAT by measuring the performance of the implemented prototype and analyze the validity of the design for parts not yet implemented. Measurements include the performance of mechanisms needed to filter components in functional stages, check components compatibility, and link them.

As threshold for a service interruption not being considered annoying by a human user, [97] considers 3 seconds. However, we presume a more flexible budget, because users have to spend additional time to physically move their attention and control from one device to another.

As testbed we use a computer that resembles hardware characteristics of everyday mobile devices. The computer is a MacBook Pro 7,1 with Intel Core 2 Duo CPU P8800 at 2.66GHz running the 64-bit Ubuntu 14.04 operating system.

The factors that affect performance when creating the search space of pipelines are the number of functional stages with compositional variability, the number of actual candidate components per stage, and the number of modalities. For this evaluation, we use the stages needed for modalities contained in two typical test multimedia streams, one for audio and one for video.

As initial repository, we use 1379 components. The functional stages used in our experiments are those defined in the application `playbin3` of GStreamer. Experiment 1 has four functional stages with compositional variability, and one

Table 8.2: Response time to create one pipeline variant

	Functional stages							Resp. time (ms) without log overhead		Num. comp.	Total queries	Repeated queries	Queries' response time (ms)	Resp. time (ms) with log overhead	
Exp.	s_1	s_2	s_3	s_4	s_5	s_6	$\|M\|$	Avg.	Stdev					Avg.	Stdev
1	✓	✓	✓	✓	N/A	N/A	1	34	2	17	111	28	50	139	4
2	✓	✓	✓	✓	✓	✓	2	280	28	27	208	112	250	449	29
3	✗	✗	✗	✗	N/A	N/A	1	9	1	17	107	8	43	36	3
4	✗	✗	✗	✗	✗	✗	2	283	17	27	207	6	161	317	35

audio modality. Experiment 2 has six functional stages with compositional variability, and audio and video modalities. To identify the time needed to filter the components per functional state, we measure the time spent to create one pipeline variant with the same topologies and number of components as Experiment 1 and 2, but no variability per stage. That is, there is only one pipeline component per stage in Experiment 3 and 4.

A `start` timestamp is recorded right after initializing the GStreamer library, setting up internal path lists, registering pipeline components, and loading standard plugins. After the pipeline is built, an `end` timestamp is recorded. The difference between `end` and `start` is the response time to build one pipeline variant with compositional variability.

As input streams we use two files; `Hydrate-Kenny_Beltrey.ogg` with audio modality for Experiment 1 and 3, and `sintel_trailer-480p.webm` with audio and video modalities for Experiment 2 and 4. We run a set of experiments starting from 10 to 1000 repetitions to observe difference in response times and standard deviation. Results from varying repetitions show differences in the order of milliseconds, which we regard as negligible.

Table 8.2 summarizes the experiments, presents the response time to create one pipeline variant, and shows how many queries are involved to create the pipeline. Query measurements are done with the tracing subsystem of GStreamer. In the table, ✓indicates that there is compositional variability in the stage, and ✗indicates the opposite. "N/A" indicates that the stage is not applicable.

Experiment 1 and 3, and Experiment 2 and 4, use the same components to build their respective pipeline. Response time in the last three columns include the time overhead spent by logging traces to obtain the query measurements. Evaluated queries are `accept-caps` and `query-caps`. The worst tracing overhead is 169 ms $(449 - 280)$ in Experiment 2. This means that queries can be answered up to 1.6 times faster than the measured time when removing the tracing overhead.

Table 8.2 shows that typical pipeline variants can be built between 9 ms and 283 ms in average, depending on the compositional variability and components

Table 8.3: Reduction of compositional variability

	s_1	s_2	s_3	s_4	s_5	s_6	β
$\|V\|$ in Exp. 1	2	1	2	2	N/A	N/A	17 ms
$\|V\|$ in Exp. 2	2	1	2	2	3	2	28 ms

involved. The time difference between Experiment 1 and 3, and the difference between Experiment 2 and 4, is the overhead caused by the compositional variability in functional stages. Experiment 4 presents the maximum average response time of 283 ms, and Experiment 2 presents the maximum standard deviation of 28 ms, both measurements have audio and video modalities. Response time is much larger for pipelines with video modality because they instantiate a *Xv*-based video sink component, which require more resources.

If we remove the instantiation time needed to initialize components with hardware dependencies, the response time to build pipeline variants in GStreamer depends on the topology and the number of components in paths due to the potential duplication of query recursion. Therefore, despite the linear complexity of the recursion queries in GStreamer, query measurements in Experiments 1-4 in Table 8.2 show that queries and the implementation of their handlers are the scaling factors when creating the variability search space.

8.3.1 Time spent to create entire search space

The design of DAMPAT allows the creation of the variant search space by incrementally changing previously created variants, such as in the MUSIC approach [93]. However, since the current implementation of DAMPAT is based on GStreamer, the needed queries and their recursion do not allow to re-utilize the operations to link connectors and verify parameters. Thus, this evaluation re-utilizes only the operations to filter components per functional stage.

Table 8.3 presents the number of resulting filtered components ($|V|$) in Experiment 1 and 2, and the time the filtering process took (β). Since, the input stream in Experiment 1 visits all stages, the maximum number of paths is 8. The input stream in Experiment 2 has two modalities; the audio modality visits stages s_1, s_2, s_3, and s_5, and the video modality visits stages s_1, s_2, s_4, and s_6. Therefore, there are 12 paths for audio, and 8 paths for video, that is a maximum number of 20 paths in total in Experiment 2.

The time spent to create the entire search space is calculated in two steps. First, we calculate the time to create all possible paths. Second, we calculate the time spent to compute all valid path combinations.

Step 1: All possible paths An approximation to the time needed to create all paths is equal to $\eta = |W| \cdot (\alpha - \beta) + \beta$, where $|W|$ is the number of all possible paths for each experiment, α is the response time spent to create one pipeline with functional stages, and β is the time spent to filter components in functional stages. Results are $\eta = 153$ ms ($|W| = 8$, $\alpha = 34$ ms, and $\beta = 17$ ms) for Experiment 1, and $\eta = 5$ s ($|W| = 20$, $\alpha = 280$ ms, and $\beta = 28$ ms) for Experiment 2.

Step 2: All valid path combinations Based on the number of filtered candidates presented in Tables 8.3, and evaluating Equation 8.1 with $k = 1$, the complete search space in Experiment 1 contains 9 path combinations, and the search space in Experiment 2 contains 273 combinations. The number of path combinations represent the maximum number of all pipeline variants.

DAMPAT uses the BRGC algorithm to enforce Equation 8.1. We evaluate the complexity of the BRGC algorithm $\mathcal{O}(2^{|W|})$, with $|W| = 8$ for Experiment 1 and $|W| = 20$ for Experiment 2. The resulting number of operations needed by BRGC are 256, and $1,048,576$. These operations can be performed in the order of microseconds in commodity hardware, such as the testbed described in this section. Therefore, we consider the time to combine all paths as negligible, and we conclude that the time spent to create the entire variability search space is in average 153 ms for audio input streams, and 5 seconds for video streams. This time is within the time budget as discussed at the beginning of this section.

8.3.2 Variant selection

The multi-dimensional utility function in Equation 8.2 maximizes the satisfaction of the user by selecting the variant with the highest utility value. The scaling performance of the variant selection depends on; 1) the complexity of the multi-dimensional utility function (Equation 8.2), 2) the complexity of each utility function per property, 3) the data structure to store the newly computed utility values of each pipeline variant, and 4) the complexity of the sorting and searching algorithm to select the variant with the highest utility.

The complexity of Equation 8.2 is linear. However, since the developers of pipeline components provide the utility function per property, DAMPAT cannot control the complexity of those functions. Typical lists as data structures, sorting algorithms with complexity $\mathcal{O}(n^2)$, and search algorithms with complexity $\mathcal{O}(n)$

do not represent an issue for the maximum variant search space from Experiment 1 and 2 (273 variants). However, if the search space is expected to increase in several orders of magnitude, other algorithms are needed. For such cases, the evaluation time of utility functions can be reduced by applying heuristics, and optimize the data structures, sort and searching algorithm.

8.4 Related work

The graph-based multimedia framework GStreamer [87] provides the mechanisms needed to build, manage and dynamically reconfigure multimedia pipelines. These mechanisms are analogue to most abstractions described in Infopipes [27]. However, since the framework is not designed to develop applications that migrate between heterogeneous devices during execution, it cannot be used out-of-the-box to design applications adhering to the application mobility paradigm.

State of the art approaches such as [170] does not discuss how to create complex graphs to represent processing of multimedia presentations in modern multimedia applications. Other adaptive systems such as [78, 93] achieve modality selection by ignoring already processed media streams, removing UI components or simply muting the audio card or switching off the display, which results in a waste of relatively significant amount of resources. Our measurements show that processing components for video and audio consume about 80% and 20% of CPU time independently, and 97% and 3% of processed data respectively.

8.5 Conclusions

In this paper we have presented the design, prototype implementation and evaluation of the planning phase of the context-aware autonomic adaptation system called DAMPAT. DAMPAT enable developers to implement mobile applications adhering to the application mobility paradigm without the burden of designing and implementing the mechanisms for autonomous context-aware adaptation of multimedia presentations. Users of applications developed in DAMPAT can move their applications during runtime to take advantage of the dynamically changing heterogeneous devices that surround them, while taking into consideration their preferences. Evaluation of DAMPAT shows that the average time to create the variability search space for typical audio and video streams is in average 153 ms, and 5 seconds respectively. This time, however, can still be eliminated by creating the search space before adaptation is needed. For this, the input stream has to be known. As future work, we propose the evaluation of DAMPAT with multiple adaptation types, and perform the planning phase entirely on a model of pipelines as in models@run.time.

Acknowledgment

This work would not be possible without valuable discussions with Prof. Thomas Plagemann, and Ellen Munthe-Kaas. We also would like to thank the GStreamer community for sharing their knowledge.

Chapter 9

P4 – SOCKMAN: Socket Migration for Multimedia Applications

Authors: Francisco Javier Velázquez-García, Håvard Stigen Andersen, Hans Vatne Hansen, Vera Goebel and Thomas Plagemann

Published in: The 12th International Conference on Telecommunications (ConTEL 2013)

Reference in Bibliography: [194]

Abstract: The dynamically changing set of multimedia capable devices in the vicinity of a user can be leveraged to create new ways of experiencing multimedia applications through migrating parts of running multimedia applications to the most suited devices. This paper addresses one of the core challenges of application migration, i.e., migration of transport protocol state that is maintained by the endpoints of established connections. Our solution fulfills the stringent temporal requirements of multimedia applications and enables migratable applications to interact with legacy applications, e.g., a migratable video player together with YouTube. The core idea of our solution, called SOCKMAN, is to provide a middleware service to hide that proxy-based forwarding is used to migrate connection endpoints, i.e. sockets, and to maintain an end-to-end perspective for the applications. The evaluation of the SOCKMAN implementation shows that SOCKMAN meets multimedia application requirements, preserves transport protocol state, and performs well on low-end devices, like mobile phones.

9.1 Introduction

The popularity of multimedia applications such as YouTube, Spotify, and Netflix has been steadily increasing during several years. Some years back multimedia applications were confined to powerful and stationary computers. However, this situation has changed, because now there are many more multimedia capable devices in the market. These devices have different capabilities related to computational power, size and quality, I/O interfaces, and mobility; and range from smart phones to desktop computers and media centers. These different devices typically serve different purposes and as such many users own and use several devices.

First steps towards an integration of these devices into one media system to leverage the device diversity can be seen in products like Apple TV. Users can redirect media streams from their smartphone, tablet, or PC to devices physically connected to Apple TV, e.g. TV screen, Hi-Fi stereo equipment. However, solutions of this kind have limitations, since they require preinstalled applications and work only for a static set of devices.

By overcoming these limitations, we envision a platform to support multimedia applications that can migrate (parts of) running applications between various devices belonging to a media system. The number of devices in the system may change dynamically, because of the mobility of users and devices. For example, a user watching a football match on a smartphone on a train is able to migrate video and audio to a home media center after coming home. In this example, the smartphone and the home media center dynamically form one media system.

Migration of (parts of) running multimedia applications requires proper solutions for process migration fulfilling the requirements of multimedia applications. Therefore, we investigate process migration in TRAMP [96] to provide a platform to develop applications that can migrate running (parts of) applications seamlessly. In this way, users of multimedia applications can benefit from a broader choice of best suited devices in their vicinity. Other process migration advantages such as install once configure once and load balancing can additionally improve the quality of experience.

In this paper we address one core problem of process migration: How to migrate transport protocol state that is maintained by the endpoints of established connections. These endpoints are typically represented in the operating system as sockets. Migration of the transport protocol state is technically challenging, because of three main reasons: (1) The widely adopted transport control protocol (TCP) and user datagram protocol (UDP) do not support mobility. Mobile IP cannot be applied in the context of process migration, because it does not transfer the transport protocol state when redirecting the IP end-to-end path to the new location of a mobile device. (2) Connection handover must meet strict deadlines of multimedia applications to avoid service interruption and not affect quality of experience. (3) Throughput must be sufficient to support multimedia traffic, if enough network bandwidth is available.

We present in this paper a solution for seamless connection handover during process migration, called SOCKMAN. SOCKMAN provides a middleware service and a proxy to migrate connection endpoints, i.e. sockets. SOCKMAN as a middleware service is portable, i.e., it can run on different hardware and operating system platforms, and the proxy hides migration of endpoints from legacy applications. SOCKMAN is a handover system that achieves socket migration using the proxy-based forwarding technique. In order to preserve the transport protocol state after socket migration, SOCKMAN transfers entire IP packets in UDP tunnels between the middleware and the proxy. This makes it possible to obey the end-to-end principle in the Internet. Our evaluation shows that SOCKMAN introduces minimal delay and consumes a moderate amount of CPU resources, even on low-end devices like smart phones. The SOCKMAN middleware and the proxy forwarding can be used without negative impact on the quality of experience for the users. The evaluation demonstrates that SOCKMAN preserves the transport protocol state after socket migration.

In order to motivate and explain the main contributions of this work, i.e. design, implementation and evaluation of SOCKMAN, we identify in Section 9.2 the requirements that SOCKMAN must fulfill for multimedia applications. Section 9.3 describes the SOCKMAN design that consists of a middleware service and a proxy. Section 9.4 evaluates SOCKMAN, and Section 9.5 studies related work. Finally, Section 9.6 presents conclusions and outlines future work.

9.2 Requirement Analysis

The requirements for connection handover systems originate from applications. Previous connection handover systems for file transfer and instant messaging applications have different requirements than a connection handover system for multimedia applications. We have identified the following five requirements for our connection handover system based on requirements for multimedia applications: (1) low handover time, (2) high throughput, (3) legacy application support, (4) portability, and (5) no special infrastructure support. In the following paragraphs, we motivate and describe each of these requirements in more detail.

Low handover time: The handover time must be sufficiently low to avoid service interruption and decreased quality of experience for the users. For example, migrating an audio application should be perceived as a seamless action when switching speakers. We characterize handover as seamless when it takes less than 100 ms, a number also used in the literature [16, 105].

High throughput: The throughput must be high enough to support multimedia traffic, provided enough network bandwidth is available. However, the throughput is not only dependent on the available network bandwidth, but also on the packet processing capabilities of the devices involved. Since we can not change hardware capabilities of devices, our software design and implementation must be efficient in order to fulfill the requirement for high throughput. We aim for throughput of at least 1.5 Mbps, because it is the recommended downstream bandwidth for Netflix [141] and Hulu [101], two popular multimedia applications for video streaming.

Legacy application support: The system must support communication with applications that are unaware of the connection handover system. We have no control over the server parts of applications such as YouTube, but users should be able to use these services. This means that our system must be able to transparently communicate with these types of applications.

Portability: The system must be portable across heterogeneous devices. Different types of devices have different software and hardware platforms. Two different mobile platforms are iOS and Android, and several operating systems exist for desktop devices, such as GNU/Linux, Windows and Mac OS. Since users might own devices running on different platforms, it is essential for our system to be portable.

No special infrastructure support: The system must work with available network infrastructure and should not require special services from Internet service providers. Since we can not change the Internet infrastructure, but the multimedia applications we target rely on it, our system has to work with current network technology and equipment.

These requirements are the foundation for the SOCKMAN design described in the next section.

9.3 Design

In this section we describe the major design decisions of SOCKMAN, present its architecture and describe a socket migration scenario. The five major design decisions concern: (a) whether SOCKMAN should use vertical or horizontal handover, (b) where it should be placed (operating system, as middleware or in the multimedia applications using it), (c) whether connection handover should be achieved with packet spoofing, host-to-host migration support or with proxy-based forwarding, (d) how transparency to legacy applications can be achieved, and (e) whether to utilize a make-before-break or break-before-make approach.

9.3.1 Vertical or Horizontal Handover

Connection handover is the process by which an active connection endpoint changes its point of attachment to a network. There are two main types of connection handover, horizontal- and vertical handover. Horizontal handover is handover using the same point of attachment to a network, e.g. a mobile phone moving between different network cells, or a laptop moving between wireless access points belonging to the same network. Horizontal handover is performed in the lower layers of the OSI model by network administrators. Vertical handover is handover using different network attachments, e.g. moving from a 802.11 network to a cellular network such as 3G, or as in our case, migrating a part of an application to a new device with another point of attachment to a network. Another reason why we create SOCKMAN as a vertical handover solution is that we do not have control over layer 1,2 and 3 in the OSI model which is required for horizontal handover.

Vertical handover solutions can be categorized as connection management or socket migration systems. Connection management systems make new endpoints for communication and reset state after migration, while socket migration systems keep state intact by moving the endpoints from one device to another. Connection management is unsuitable for SOCKMAN, because it requires modification in both ends of communication channels, breaking the requirement for legacy application support. Socket migration does not rely on support from network operators or Internet service providers.

9.3.2 Placement of SOCKMAN

There are different alternatives where SOCKMAN can be placed, in the operating system as a kernel module, as middleware or as a part of a multimedia application.

Connection handover can be accomplished in any of these, but since we have the portability requirement and want to be able to use SOCKMAN on heterogeneous devices, we can not design SOCKMAN as a kernel module. Placing SOCKMAN as an integrated part of multimedia applications can severely limit its potential. A general service that can be used by many applications has larger potential, and can be tested and verified once and used by any type of application. This leaves middleware as the most suitable placement for SOCKMAN.

9.3.3 Connection Handover Technique

According to Kuntz and Rajan [117], three techniques for socket migration exist: packet spoofing, host-to-host migration support and proxy-based forwarding.

Packet spoofing takes place when a client sends packets with a forged source address and port number. The server sends replies back to the forged address, and the client intercepts the replies. This can enable a migrated client application to preserve a connection by pretending it never moved. Examples of systems using this technique are SockMi [24] and Netfilter live migration [99]. Packet spoofing is not suitable in our scenario, because it only works in un-switched networks where devices are able to see each others packets.

Host-to-host migration happens when a client sends its new IP address and port number to the server, and the server updates its endpoint correspondingly. This connection handover technique is used in MIGSOCK [117], Reliable Sockets (Rocks) [207], and Migratory TCP (M-TCP) [179]. Host-to-host migration is not suitable in our scenario, because it requires both ends of the communication channel to be migration-aware, breaking the requirement for legacy application support.

Proxy-based forwarding is illustrated in Figure 9.1. The figure shows a scenario where a proxy acts on behalf of a multimedia application. Using a proxy allows the multimedia application to migrate without informing the legacy application. Proxy-based forwarding systems do not need any special infrastructure support and can support legacy applications. This means that a proxy-based forwarding solution fulfills our needs without breaking any of the identified requirements. An overview of related work in proxy-based forwarding systems is provided in Section 9.5.

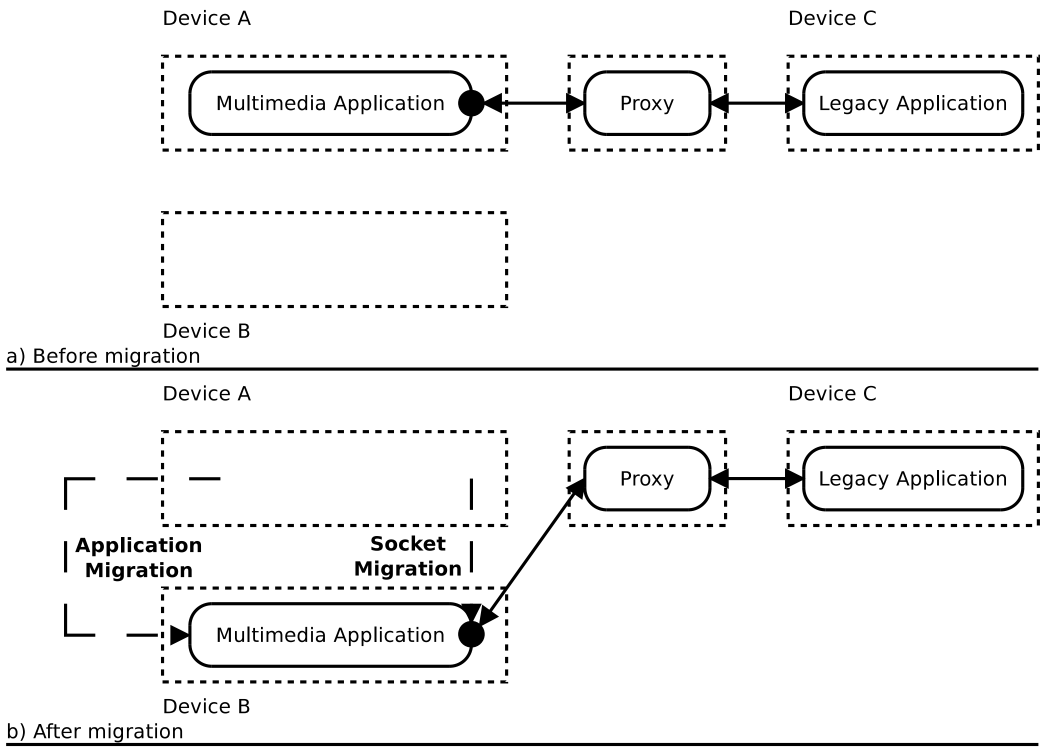


Figure 9.1: Socket migration using proxy-based forwarding.

9.3.4 Legacy Application Support

The connections between the multimedia applications and the proxy must be hidden from the legacy applications in order to support the envisioned application mobility. One option to achieve this transparency is to use two distinct connections: one from the multimedia applications to the proxy and another from the proxy to the legacy applications. This option duplicates the connection control overhead and breaks the end-to-end principle. A more suitable option is to tunnel packets between the multimedia applications and the legacy applications using the proxy only as a forwarder. The multimedia application uses the middleware to connect to the legacy application through the proxy. The proxy does not alter the packets going through it. In this way, the endpoints can handle all connection management and the proxy can use the socket state maintained by the multimedia applications. We choose to tunnel packets from the multimedia applications in UDP packets, because it avoids the double reliability of sending TCP packets inside TCP packets. In other words, the payload of the UDP packets are entire transport layer packets, i.e. UDP/IP or TCP/IP packets, comprising packet header and payload. In this way, completeness and correctness of data is verified only in the endpoints and not in the proxy.

SOCKMAN requires all connections to be initiated by the multimedia applications. This one-way connection establishment is required because the proxy needs a forwarding table in order to know where to forward packets coming from the legacy applications. However, this is not a limitation for clients of multimedia applications, because client-server connections are initiated by clients. Figure 9.2 shows a socket migration scenario using proxy-based forwarding and tunneling of packets.

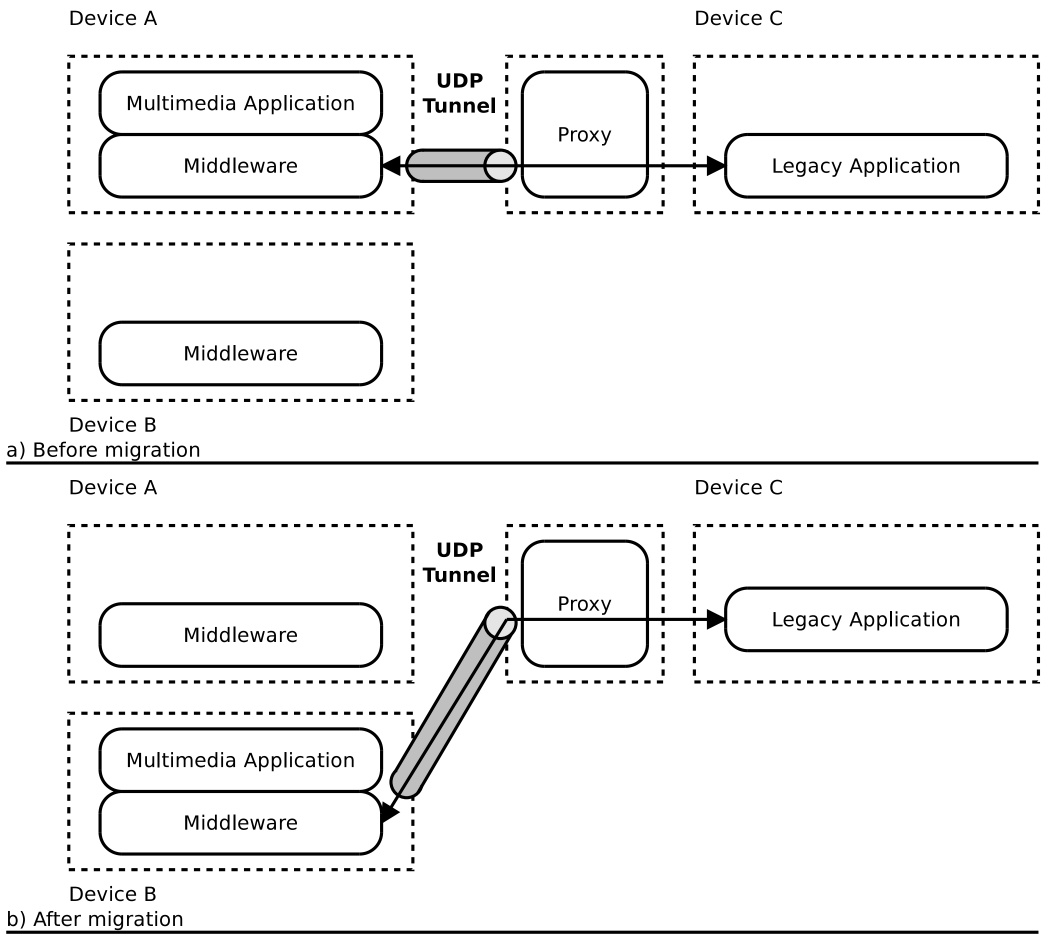


Figure 9.2: Migration scenario using proxy-based forwarding and UDP/IP tunnels.

9.3.5 Connection (Re-)establishment

Our final design decision concerns the sequence of connection establishment and tear-down, where break-before-make and make-before-break are two alternative techniques [138]. Systems using make-before-break establish new connections before they tear down old connections, and correspondingly, systems using break-before-make have only one connection active at a time. The make-before-break technique can achieve less packet loss and delay than the break-before-make approach, but it requires a more complex design and implementation. SOCKMAN uses the break-before-make technique, because of its simplicity, and because it can fulfill all requirements for low connection handover time.

9.3.6 Architecture

The SOCKMAN architecture consists of two main components and four internal modules. The two main components are the middleware and the proxy. It is possible to separate the middleware and proxy components, but they are integrated in SOCKMAN, because they share several functions, described below.

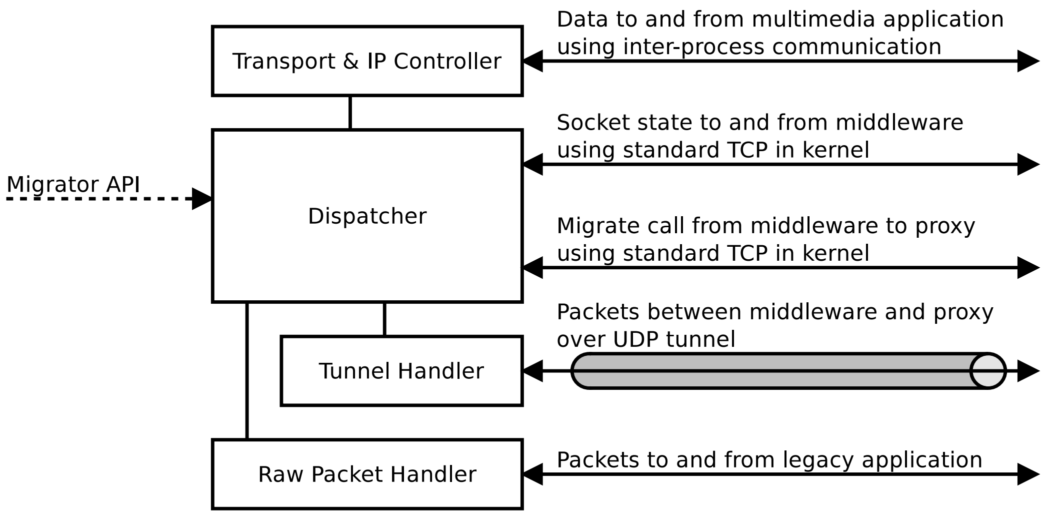

Transport & IP Controller
Data to and from multimedia application using inter-process communication
Socket state to and from middleware using standard TCP in kernel
Migrator API
Dispatcher
Migrate call from middleware to proxy using standard TCP in kernel
Packets between middleware and proxy over UDP tunnel
Tunnel Handler
Raw Packet Handler
Packets to and from legacy application

Figure 9.3: The SOCKMAN architecture consisting of four modules.

Figure 9.3 shows the SOCKMAN architecture and data flow paths. We have separated the concerns of SOCKMAN into four minimal and efficient modules:

The `Transport & IP Controller` module provides an Application Program Interface (API) similar to POSIX sockets where multimedia applications can use functions like `send()` and `receive()` to communicate with legacy applications. The component currently contains UDP and basic TCP functionality, but any transport protocol can be placed in this controller. When data from a multimedia application comes into the controller, appropriate headers are added depending on the transport protocol and current socket state (sequence numbers etc). After the headers are added, the packets are sent to the dispatcher. Packets coming in the opposite direction are delivered to the multimedia application.

The `Dispatcher` module handles internal data flow by passing data between components. The dispatcher receives a function call from the external migrator component about when and where to migrate sockets. It notifies the proxy about when and where sockets migrate using the standard TCP implementation in kernel. In addition, it sends and receives the state information necessary to rebuild sockets. Socket state is also exchanged using the standard TCP implementation in kernel. The size of a UDP socket state is 32 bytes and the size of a TCP socket state is 96 bytes so it always fits in one packet.

The `Tunnel Handler` module is responsible for the data sent between the middleware and the proxy. It maintains UDP tunnels, shown in Figure 9.2, and sends the encapsulated packets created by the `Transport & IP Controller`. Since UDP connections are stateless, the proxy does not need to handle packet loss and hence consumes little CPU resources.

The `Raw Packet Handler` module is used by the proxy to send and receive IP packets to and from the legacy applications.

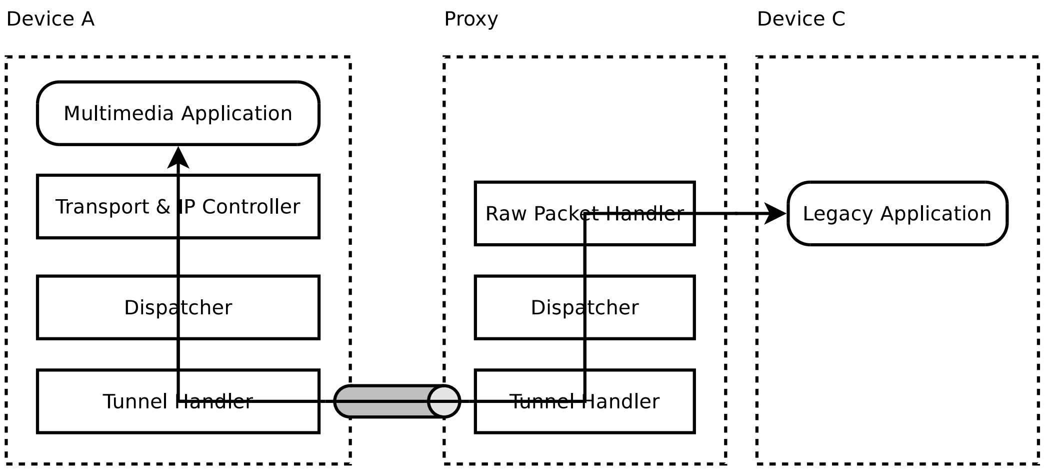


Figure 9.4: Data flow path in SOCKMAN using three devices.

Figure 9.4 shows how data flows between the SOCKMAN modules. In one direction, the middleware sends data from the multimedia application through the `Transport & IP Controller`, the `Dispatcher`, and the `Tunnel Handler`. The `Tunnel Handler` sends the data to the proxy on another device where the `Tunnel Handler` receives it and sends it to the `Dispatcher`, `Raw Packet Handler`, and in turn to the legacy application on device C. In the opposite direction, the data flows from the legacy application to the multimedia application in reverse order.

An external migrator module calls a function in SOCKMAN informing it about when and where to migrate sockets. It is not a part of SOCKMAN, but uses an open API to initiate socket migration. The migrator performs checkpointing of application state (excluding socket state), sends and receives application state, and stops and starts applications. This module is currently subject to ongoing research in the TRAMP project.

To summarize, the middleware uses the `Transport & IP Controller`, the `Dispatcher` and the `Tunnel Handler`, while the proxy uses the `Tunnel Handler`, the `Dispatcher` and the `Raw Packet Handler`. The migrator is an external component using the SOCKMAN API to initiate socket migration.

9.3.7 Socket Migration Scenario

Figure 9.2 illustrates a socket migration scenario in SOCKMAN, with four devices involved. The multimedia application is running on device A and is migrated to device B. The legacy application is running on device C. The middleware is running on device A and device B. The proxy is running on a dedicated device, but in other scenarios it is possible that the proxy runs on device A or device B. The multimedia application is communicating with a legacy application using TCP. The socket migration process starts when a user wants to migrate a multimedia application from device A to device B. An external migrator is used to stop, send, and resume the application. In addition, the migrator uses the SOCKMAN API to inform the `Dispatcher` where it is migrating to, so that the proxy can be updated. Figure 9.5 shows the messages being sent between the devices involved.

Data is being continuously sent from the legacy application running on device C to the multimedia application running on device A, via the proxy. For simplicity, we consider only unidirectional communication in this scenario, but bidirectional communication is supported in SOCKMAN. Device A acknowledges the data.

The user decides to migrate the multimedia application to device B, and the

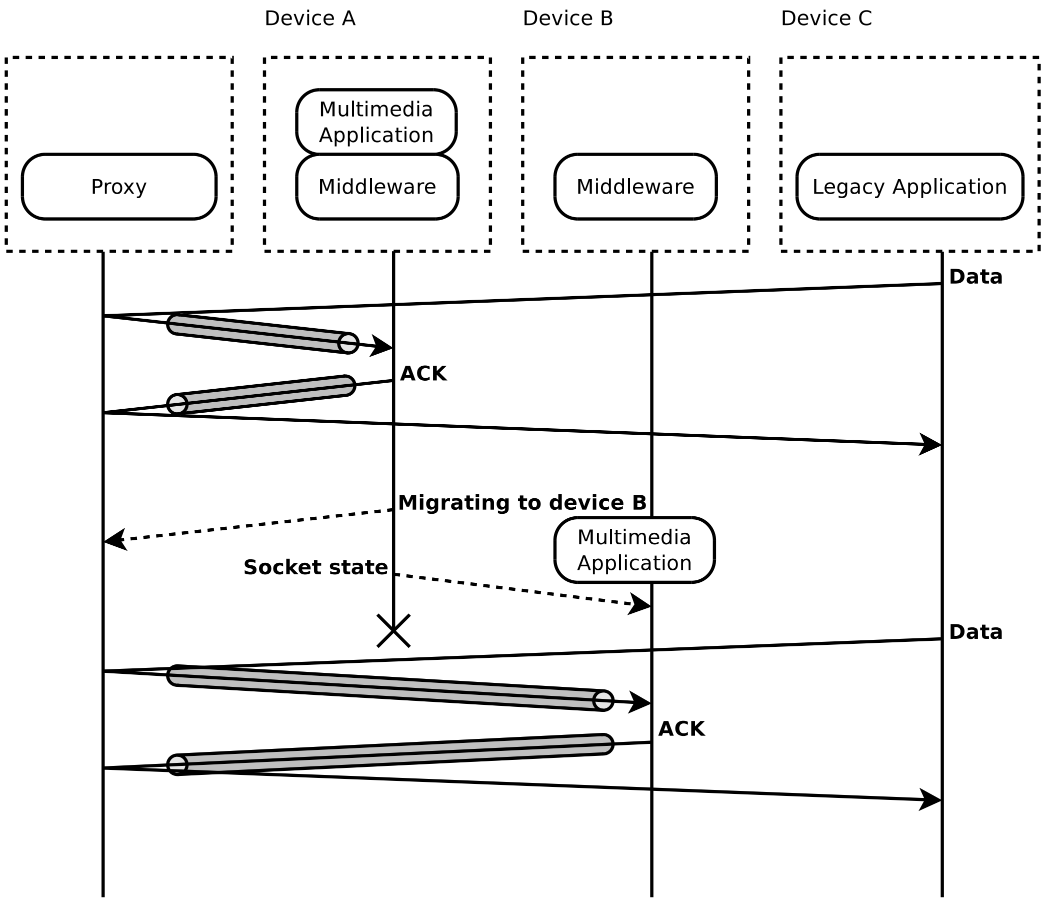


Figure 9.5: Message passing during socket migration. Fixed lines illustrate regular traffic, gray background illustrates tunneled traffic, and dotted lines illustrate migrate call and socket state sent using the standard TCP implementation in kernel.

migrator uses the API, provided by the `Dispatcher`, to initiate migration. The `Dispatcher` establishes two dedicated connections using the standard TCP implementation in kernel, one to device B and one to the proxy. The middleware uses these two connections to inform the proxy about the IP address and port number of device B, and to send the socket state to device B. The socket state consists of the IP addresses and port numbers, sequence numbers, and a buffer of unacknowledged outgoing packets. Unacknowledged incoming packets that are lost during migration are retransmitted by the sender when using a reliable transport protocol.

SOCKMAN starts buffering data from the proxy after the socket is rebuilt on device B. Once the application resumes, SOCKMAN flushes the buffered data and end-to-end communication resumes. The data is now tunneled via the proxy to device B, which has taken over the transmission control responsibility. Thus, data is received and acknowledged.

SOCKMAN is implemented using the C programming language. It is compiled and tested on GNU/Linux, and can be compiled on Mac OS with only minor modifications. However, because of limitations on raw sockets in Windows, compiling SOCKMAN on Windows requires more work. For a more detailed description about the SOCKMAN design and implementation, we refer to [9].

9.4 Evaluation

In this section we evaluate SOCKMAN to show that it fulfills the multimedia requirements. We use the following metrics: socket migration time, latency overhead, throughput and CPU load. The workload consists of a streaming server representing the legacy application, streaming data with 100 bytes payload to a multimedia application over UDP. The testbed consists of one low-end device and three high-end devices. The low-end device has an Intel Atom N270 CPU with 1.6 GHz and 2 virtual cores, 2 GB of RAM and a 100 Mbps network interface. It is running the 32 bit version of Ubuntu Linux 11.11. The high-end devices have Intel Core i7 CPUs with 2.93 GHz and 8 virtual cores, 4 GB of RAM and a 1 Gbps network interface. They are running the 64 bit version of Ubuntu Linux 11.04. All devices are located in the same local area network.

9.4.1 Socket Migration Time

Socket migration time is the time it takes from starting to export the socket state of a multimedia application t_s, to the time the socket is reinstated on a different device t_r, calculated as $socket\_migration\_time = t_r - t_s$. We determine t_s and t_r by migrating a socket using UDP between two high-end devices, device A and device B in Figure 9.2. The proxy is running on the low-end device, and the legacy application is running on another high-end device. The external migrator component is emulated and interaction with the Migration API is done via the command-line interface. Device A exports the socket state, informs the proxy about the IP address and port of the new device, and sends the socket state to device B. This results in two packets being sent from device A, one to the proxy and one to device B. Device B imports the socket state. Both t_s and t_r are registered in device A, t_s when the Migration API receives a migrate call, and t_r when device

B has confirmed that the socket is rebuilt. We repeat the experiment 12 times and measure the average socket migration time in this setup to be 0.593 ms with a standard deviation of 0.061 ms. This time is clearly below our limit of 100 ms for connection handover, which is tolerable by most users of multimedia applications [16, 105].

Table 9.1: Average data loss *DL* in kilobytes, minimum number of lost packets *PL*, and probability *P* of an additional packet loss for four migration times and three bitrates.

Socket Migration Time	0.593 ms (measured)			1 ms (LAN)			10 ms (MAN)			100 ms (WAN)		
Bitrate (Stream)	DL	PL	P	DL	PL	P	DL	PL	P	DL	PL	P
192 kbps (MP3)	0.014	0	0.013	0.024	0	0.020	0.240	0	0.167	2.400	1	0.642
1.5 Mbps (Netflix)	0.114	0	0.106	0.192	0	0.159	1.920	1	0.338	19.200	13	0.135
9.8 Mbps (DVD)	0.744	0	0.689	1.254	1	0.038	12.544	8	0.745	125.440	85	0.815

The physical location of the proxy influences the socket migration time. High latency between the middleware device and the proxy device will give a high socket migration time. This is because the migration call must be sent from middleware to proxy, as illustrated in Figure 9.5. Therefore, the placement of the proxy is critical to achieve a low socket migration time.

In order to get an intuition about the potential packet loss during socket migration, we present Table 9.1 with calculated numbers for four migration times and three bitrates. The metrics in the table are the average data loss *DL* in kilobytes, the minimum number of lost packets *PL* and the probability *P* of an additional packet loss. The migration time of 0.593 ms is given by our measurements, while 1 ms, 10 ms and 100 ms are estimated socket migration times in typical LAN, MAN and WAN networks respectively. The bitrates in the table are examples for typical multimedia applications.

9.4.2 Latency Overhead

The latency overhead is the time difference between a direct data transfer from the multimedia application to the legacy application l_d and the same data transfer through the proxy l_p, calculated as $latency\_overhead = l_p - l_d$. We determine l_d and l_p by comparing the network latency with and without the proxy. The multimedia application and legacy application run on two high-end devices and the proxy runs on the low-end device. The ping tool is used to determine the network latency. It is done by summing the latency between the multimedia application and the proxy, and between the proxy and the legacy application. This addition of two link latencies is emulating a one-hop link. We compare this with an end-to-end transmission between the multimedia application and the legacy application over the proxy. The results show that using a proxy does introduce latency overhead, but that the overhead introduced in our scenario is negligible for multimedia applications. In our scenario the proxy overhead when running on the low-end device is 0.175 ms. Larger overheads can occur if the proxy is far away from the shortest path between the multimedia application and the legacy application.

9.4.3 Throughput

To measure the throughput of SOCKMAN, we run the multimedia application and the legacy application on two high-end devices, and the proxy on the low-end device. Packets are sent over UDP in both directions to verify that this has no impact on performance. We determine that the proxy is the limiting factor in the experiment and that it can process up to 20000 packets per second before packets are dropped. 20000 packets per second with 100 bytes payload is over 15 Mbps, which is more than the requirement of 1.5 Mbps. To determine the limits of SOCKMAN we increase the payload from 100 bytes to 1400 bytes, which is close to the maximum transmission unit of the network. With this payload, we are able to achieve full network utilization of 1 Gbps, meaning that SOCKMAN is able to utilize the full network capacity if the payload and packet size is appropriate, and that the requirement for throughput is met even on resource constrained devices.

To gain insights on how SOCKMAN can perform using a kernel implementation of TCP we compare the SOCKMAN TCP implementation in user space with a kernel implementation of TCP. Measurements are performed in the low-end device. To simulate a realistic scenario, a 5,460 kB file is sent over a typical residential WLAN (802.11g). We assume that the actual throughput for the file transfer will be substantially lower than 54 Mbps (the maximum theoretical throughput in 802.11g networks) due to wireless interference from nearby networks, competing flows, and CPU utilization in sending or receiving devices affect the achievable throughput. In addition, TCP window size and round trip time impact bandwidth utilization in TCP connections [4]. The measurements show that the SOCKMAN TCP implementation achieves an average throughput of 5.41 Mbps with a standard deviation of 1.237 Mbps. The kernel implementation achieves a higher average throughput of 19.593 Mbps with a standard deviation of 0.852 Mbps. One reason for this difference is the lack of TCP options, such as window scaling and TCP timestamps, in the SOCKMAN TCP implementation.

9.4.4 CPU Load

In order to measure the CPU load of the SOCKMAN middleware and proxy, and the SOCKMAN TCP implementation in user-space, separate experiments are done on the low-end device using the top tool.

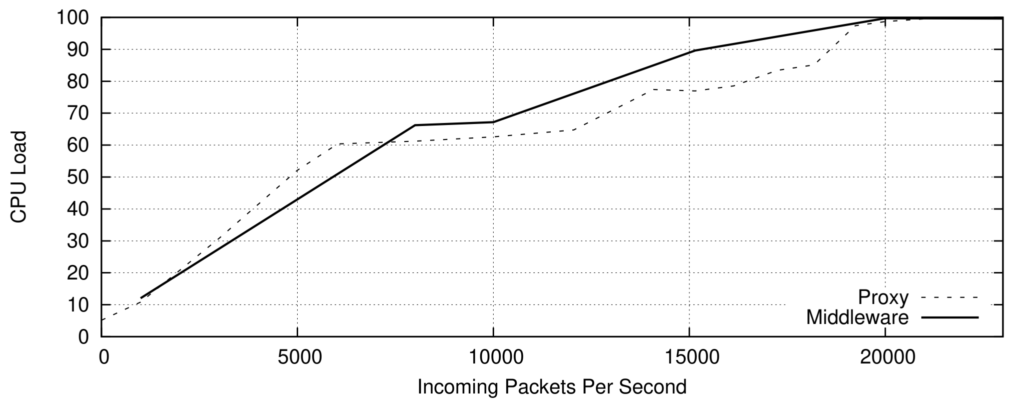


Figure 9.6: CPU load of the SOCKMAN middleware and proxy running on the low-end device.

The CPU load of the SOCKMAN middleware and proxy is visualized in Figure 9.6. It shows that both the middleware and proxy can process up to 20000 packets per second without exhausting the CPU, meaning that it is possible to run SOCKMAN on resource constrained devices, such as smart phones.

The CPU load of the SOCKMAN implementation of TCP is measured in two experiments. First, a large file is sent from the legacy application to the multimedia application without controlling the bit-rate. In this experiment, the SOCKMAN implementation of TCP has an average CPU load of 138% (where 200% is maximum because of the two virtual cores in the low-end device) with a standard deviation of 3.916%, while the kernel implementation has an average CPU load of 4% with a standard deviation of 0%. One reason for the higher CPU utilization in this experiment is busy waiting when the TCP window is full. This situation can be avoided by performing a blocking operation instead. Second, a 1.5 Mbps stream is sent from the legacy application to the multimedia application. In this experiment, the SOCKMAN implementation of TCP has an average CPU load of 9%, while the kernel implementation has an average CPU load of 1%.

9.4.5 Summary

The results from the evaluation show that SOCKMAN fulfills the following requirements for multimedia applications: short socket migration time, high throughput, and low latency. The results from the CPU load experiment show that SOCKMAN can run on devices with limited processing capabilities. Using SOCKMAN, multimedia applications can migrate their sockets and continue communication with legacy applications without decreased quality of experience for the users.

9.5 Related Work

Table 9.2: State-of-the-art – N/A means that we do not have enough information to determine whether a requirement is fulfilled or not, PBF means proxy-based forwarding, PS means packet spoofing, and HHMS means host-to-host migration support.

System	Handover Time	Throughput	Legacy Applications	Portability	No Special Infrastructure Support	Category	Handover Technique
SMP [127]	No	Yes	Yes	No	Yes	PBF	Connection Management
Zap [147]	No	Yes	Yes	No	Yes	PBF	Socket Migration
UPMT [31, 30]	Yes	Yes	Yes	No	Yes	PBF	Connection Management
MSOCKS [137]	No	Yes	Yes	No	Yes	PBF	Connection Management
SockMi [24]	N/A	N/A	Yes	No	Yes	PS	Socket Migration
Netfilter Live migration [99]	N/A	No	Yes	No	Yes	PS	Socket Migration
MIGSOCK [117]	Yes	Yes	No	No	Yes	HHMS	Socket Migration
Rocks [207]	Yes	Yes	No	Yes	Yes	HHMS	Socket Migration
TCP-R [71]	Yes	Yes	No	No	Yes	HHMS	Connection Management

Related work is summarized and compared in Table 9.2. This table shows state-of-the-art systems with respect to the requirements identified in Section 9.2, their category and their connection handover technique. First, we analyze insights from Table 9.2, and later we detail key differences between proxy-based forwarding systems, SMP [127], Zap [147], UPMT [30, 31], MSOCKS [137], and SOCKMAN.

Table 9.2 shows that all systems do not require special infrastructure support. Proxy-based systems can achieve enough throughput, support legacy applications, but do not support portability. Packet spoofing solutions support legacy applications but not portability. Host-to-host migration support systems always meet handover time and throughput requirements, but none support legacy applications.

SMP [127] is a proxy-based forwarding system that aims to avoid frame loss in MPEG4 streaming applications. It uses two connections, one between the streaming server and the proxy, and another between the proxy and the client. SMP does not migrate transport protocol state, but establishes a new TCP connection between the proxy and the client after migration. It maintains end-to-end connections by modifying TCP packets in the proxy. SOCKMAN avoids this type of processing overhead in the proxy by forwarding unmodified IP packets. Additional overhead is introduced in SMP by using a MySQL database to keep track of application locations. The authors claim that this database introduces an overhead of 200 ms. The NS2 evaluation of SMP shows that the total migration time is in the order of seconds. This delay does not meet our handover time requirement.

Zap [147] migrates groups of processes, which decouples processes from dependencies of host operating systems. Processes using Zap connect through virtual addresses, which are mapped to physical addresses. In that way, migrated applications continue using the same virtual address before and after migration. A proxy maintains the mapping, between the virtual- and the physical address of processes. Zap breaks the Internet end-to-end principle because the proxy maintains session connectivity by modifying TCP sequence and acknowledgment numbers. In SOCKMAN, the end nodes are responsible for maintaining session connectivity, i.e. transport protocol state. Evaluation of Zap shows that migrating a 363 kilobyte telnet application takes a disproportional amount of time considering its small size. This is because Zap sends a message to the remote end of the connection to inform it about the new location of telnet, and because the remote end of the connection must set up address translation rules. On migration request, a TCP connection must be established between the proxy and the device where processes are migrated to. This introduces overhead of the time spent by the three-way TCP handshaking during connection establishment. SOCKMAN avoids this overhead by using connectionless UDP tunnels between the middleware and the proxy.

UPMT [31, 30] achieves connection handover for applications running on multihomed devices. If a user of a mobile phone loses wireless connectivity, UPMT can hand over connections to the cellular network. Since UPMT does not migrate applications, it does not transfer the transport protocol state to other devices. Moreover, UPMT is composed by components in user- and kernel space, this does not comply with the portability requirement. The SOCKMAN design is influenced by the IP in UDP tunnels technique used in UPMT.

MSOCKS [137] is a solution for connection management of multihomed devices. MSOCKS does not migrate transport protocol state, but creates a new TCP connection between the proxy and the new point of attachment to the network. As described for SMP and Zap, the proxy maintains session connectivity, manages the end-to-end connection by modifying TCP headers. According to the authors, the overhead is minimal, because it is done in the kernel. Lastly, MSOCKS limits applications to use only TCP connections.

Therefore, to the best of our knowledge, no connections handover systems fulfill all our requirements. SOCKMAN has the following advantages over the presented systems: (1) It migrates transport protocol state, (2) it introduces minimal overhead in the proxy, (3) it avoids connection establishment overhead by using connectionless UDP tunnels, (4) it supports different types of transport protocols, and (5) it hides socket migration from legacy applications.

9.6 Conclusions

Users can experience multimedia applications in new ways if parts of the applications can migrate seamlessly to the most suitable devices in the vicinity of a user, e.g. moving video to a big-screen TV and audio to a Hi-Fi stereo system. In this paper, we address one core challenge of this type of migration, i.e. the migration of connection end-points, by designing, implementing and evaluating a connection handover system called SOCKMAN. The core idea of SOCKMAN is to use a proxy-based forwarding technique to forward entire IP packets in tunnels, hiding migration from legacy applications. The proxy forwards IP packets without modifying them. This preserves the transport protocol state and obeys the Internet end-to-end principle.

The evaluation shows that SOCKMAN fulfills the multimedia requirements of low handover time, high enough throughput, and legacy application support. In our experiments, SOCKMAN achieves an average socket migration time of 0.593 ms, a maximum latency overhead in the proxy of 0.271 ms, and full network utilization of 1000 Mbps when using appropriate packet sizes. In addition, the evaluation shows that SOCKMAN is able to run on resource constrained devices, such as smart-phones, without exhausting their CPU resources.

Open issues include rewriting the SOCKMAN API to accurately resemble the POSIX equivalent functions, extending the one-way initiation with support for two-way connection establishment, and implementation specific improvements like removing busy waiting to improve performance. Future work includes support for NAT traversal, design of a proxy placement algorithm, and managing security issues such as authentication, authorization and encryption.

9.7 Acknowledgment

The authors would like to thank Ellen Munthe-Kaas, Piotr Srebrny and the anonymous reviewers from ConTEL 2013 for valuable feedback.

Chapter 10

P5 – Efficient Data Sharing for Multi-device Multimedia Applications

Authors: Hans Vatne Hansen, Francisco Javier Velázquez-García, Vera Goebel, Thomas Plagemann

Published in: Proceedings of the Workshop on Multi-device App Middleware 2012

Reference in Bibliography: [95]

Abstract: By utilizing the complementary advantages in screen size, network speed and processing power, the computing devices we own can work together and provide a better user experience. By separating the concerns of an application into components responsible for distinct tasks, these components can run on the different devices where they perform best. As a step towards multi-device applications, we have designed, implemented and tested a collaboration platform for application data sharing, optimized for low producer-to-consumer delay. Distribution trees are built automatically by our system based on latency, and the total producer-to-consumer delays measured in our experiments are below the delay requirements for multimedia applications.

10.1 Introduction

Mark Weiser's vision of ubiquitous computing [196] has become true in the quantitative aspect. Many of us are surrounded by computing devices like laptops, smart-phones, tablets and home media centers. However, the qualitative part of Weiser's vision regarding seamless and unconscious collaboration between devices is unfulfilled. We aim to come one step closer to this vision by enabling multi-device multimedia applications. To maximize the Quality of Experience, applications should be able to leverage the complementary properties of the surrounding devices, like screen size, network speed and processing power. For example, we want to use the largest screen nearby when having a video-conference. However, the available devices might change during run-time. It is necessary to develop a solution that allows running parts of an application on different devices at different times. This requires to separate the concerns of an application into components such that each component is responsible for a distinct task. Furthermore, a collaboration platform is needed to enable the components of a single application that run on different devices to efficiently cooperate. In the future, the components should not be bound to one device, but be able to migrate depending on context and user preferences.

Low latency is a crucial requirement for a collaboration platform because we want to use it for multimedia applications. Real-time traffic, such as video-conferencing, multi-player network games and user-interface actions require data transfer latency between 100 and 200 ms, depending on the type of application [16, 160, 21]. We aim to support all of these types of applications and target data transfer with a latency of less than 100 ms.

It is also important that the provided collaboration platform and API is easy to use. Application developers should focus on making great applications, not on underlying data propagation and component cooperation. The natural way for programmers to access data is through memory, and an approximation to this standard paradigm is needed.

We address the challenges related to a collaboration platform, because it is the foundation for multi-device, multimedia applications. Our solution that fulfills the multimedia application domain requirements is described, and we present the following contributions: (1) A component model for fine-grained applications. (2) Design and implementation of the collaboration platform and its location transparent API. (3) Evaluation of the platform's performance with respect to the multimedia application domain.

Results from measurements with 15 real devices are evaluated to analyze the producer-to-consumer latencies and overhead of our system. Further issues, like discovery of devices, device selection, trust relations between devices and migration of running components are subject to ongoing work and out of the [sic] scope of this paper. In Figure 10.1 we show an example of how a video-conferencing application can benefit from our framework.

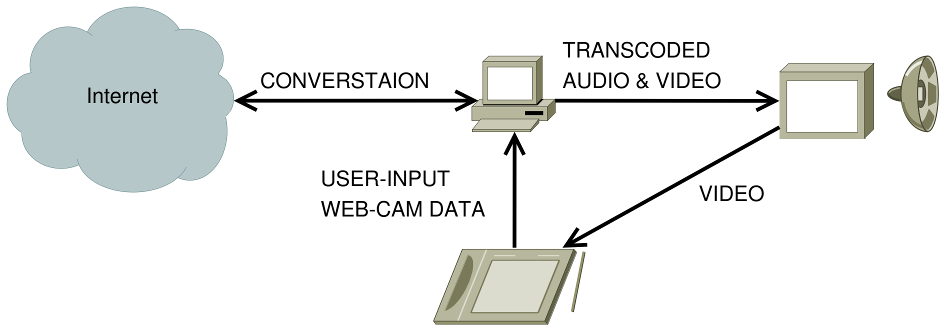


Figure 10.1: Separation of concerns and data flow in an example video-conferencing application.

A permanently connected desktop computer without power constraints communicates with all conference participants through the Internet, ensuring a reliable connection and real-time transcoding of incoming and outgoing data streams. Audio and video is played on a hi-fi equipped media center with a large screen attached for optimal quality of experience. The user interacts with an easy to use touchscreen tablet, and the tablet can also display small video thumbnails of all the participants in the conversation.

The remainder of the paper is structured as follows: Section 10.2 gives an overview of related work. Sections 10.3 and 10.4 describe the design and implementation of our collaboration platform, while Section 10.5 presents our evaluation and results. Section 10.6 concludes the paper and outlines future work.

10.2 Related Work

Distributed Shared Memory (DSM) is a data communication abstraction where memory segments are shared amongst a set of devices. This abstraction can simplify the programming of distributed components. Several DSM systems have been proposed, and much research has gone into minimizing network traffic and reducing latency between components in DSM systems. One possible optimization is to share only the data structures that need to be used by more than one component. This strategy reduces bandwidth by minimizing the amount of data that must be shared. Another optimization is to replicate the shared data on multiple devices, minimizing the latency when locating and retrieving data [180].

Linda [75] is one coordination model for DSM. It allows sharing of passive data tuples, like ("foo", 1337, 42), through a simple API. The main drawback of Linda-based systems is that they trade performance for consistency by using blocking operations. In addition, most of the Linda-based systems, like JavaSpaces [69] and TSpaces [124], are statically centralized with one node responsible for all data. This is not suitable for our collaboration platform because of the inherent churn in personal device federations.

DSM has also been used in ad hoc mobile environments, similar to our personal device federations, where devices can join and leave at any time. One approach for handling this churn is to dynamically distribute the data. Lime (Linda in a Mobile Environment) is such a system, where components make their data available to other currently connected components [156]. However, since each data structure is only managed by one device, devices with popular data can become overloaded and suffer from throughput and latency impairments.

SPREAD (Spatial PRogramming Environment for Ambient computing Design) [49] follows a similar approach where components can write to their own data segments, but read data segments from all components running on connected devices. The main drawback of SPREAD is that it uses a read-driven strategy where data is only propagated to consumers on explicit request. This uses less bandwidth since data is only sent to devices when needed, but increases the latency because of the blocking read request. This approach is not suitable for real-time, multimedia data.

Munin [23, 42] is a shared variable system utilizing multiple consistency protocols tailored to different types of shared data. Shared data in Munin is annotated with an access pattern and the system tries to choose the optimal consistency protocol suited to that pattern. The access pattern that is most similar to our collaboration platform is the producer-consumer pattern, where data is produced by one thread and consumed by a fixed set of other threads. Munin uses release consistency to minimize the number of required messages to keep data consistent. Release consistency postpones propagation of data until a release is performed allowing updates to be queued at the expense of higher latency. Buffering is acceptable for certain types of parallel programs, but not for multimedia applications.

The system most similar to our collaboration platform is presented in [47]. Corradi et al. use a hierarchical data distribution model where data is only accessible within a certain scope. Like our solution, they build distribution trees, but the way the trees are built differs. Their solution attempts to define a scalable replication scheme with a limited scope, while we focus on performance, rather than scalability. Corradi et al. separate "execution nodes" (producers/consumers) from "memory nodes" (replicators) and try to optimize the replication in order to minimize coordination efforts, but still have data coherence. For our small federations of personal devices, a distinction between execution nodes and memory nodes seem unrealistic as we do not have enough available devices.

We have seen that decoupling in time and space is provided by several DSM systems, but that none of the reviewed systems can provide the low latency we need for multimedia applications.

10.3 Design

We have designed a collaboration platform where application components can share data segments in a location transparent fashion within a small federation of personal devices. Distribution trees are created for each individual data segment and optimized for low producer-to-consumer delay. The separation of concerns for our system, and the derived design choices, are presented in this section.

A component is one distinct part of an application, and different components can work together to form a complete application, such as the video-conferencing example in Figure 10.1. Components can run locally on one machine or be distributed over a network, and in order to cooperate and function as one, components need to share data with each other. There are no restrictions on what a component can do or what type of data it produces, consumes or shares in our system. This is decided by the application component developers. However, the design is tailored for applications with real-time requirements. Data is exchanged using a flexible DSM solution where producers associate identifiers called labels with data they wish to share. As an example, a 100 ms buffer of a video stream can be associated with the label *webcam_buffer*. The size of data segments can range from bytes to megabytes depending on the individual application components. In the video-conferencing example, we know that the bit-rate for most commonly available MPEG-2 encoded video assets is 3.75 Mbps [160]. A 100 ms buffer in this scenario needs 48 kB. Smaller data segments, such as a signal from a remote control, can be as small as 1 byte. Data can be shared in two ways using our system. First, it is possible to use one label for each data segment in the application component. This is practical when different consumers are interested in different parts of the producing component's data. The other approach is to publish one large data segment representing everything a component produces. For example, metadata like a frame counter can occupy the first 4 bytes, while the remaining bytes can contain the actual frame.

Typical users own a small set of devices that need to cooperate with each other in order to achieve ubiquitous computing [53]. Our collaboration platform creates a federation of a user's devices by forming a peer-to-peer overlay. This overlay has a full mesh topology. A full mesh has direct connections between all devices and yields high performance. Albeit this topology does not scale, it suits small federations of personal devices. Trust in the users' federations can be achieved by simple security mechanisms like pre-shared keys, but the details of these techniques are out of scope for this paper.

Another concern is how the distributed components communicate. Producing components do not need to know how many consumers they have, nor where the consumers are located. Location transparency is needed, and the coordination effort must be abstracted away from the individual components. The two main communication paradigms in concurrent computing are message passing and distributed shared memory. While message passing is more flexible, only DSM can provide decoupling in time and space. We use DSM because of the need for loose coupling. Russello et al. classify DSM systems using the following set of categories [164]: Statically centralized, fully replicated, statically distributed, dynamically centralized and structurally replicated.

Our collaboration platform utilizes a structurally replicated schema where consumers have two roles. In addition to being consumers, they also act as replicators of data. The distribution trees are built based on latency, one tree for each data segment. This mechanism provides redundancy and full decoupling between producers and consumers. Full memory space sharing is not possible because our collaboration platform is intended to run on heterogeneous devices, so we focus on shared data segments.

Concurrency is an issue in shared data systems when several producers want to write to the same data segment. The standard solution to this problem is to introduce locks, which in turn leads to blocking. This is a suboptimal solution for real-time data. We allow only one producer per data segment as an alternative to locks. This avoids blocking, but application developers need to be aware of this in advance and implement their components accordingly, e.g., by adding a dynamic suffix to the label. Label description and discovery is out of the scope of this paper.

Our collaboration platform can be used for all types of distributed applications, but the distribution system is optimized for multimedia applications. As latency is crucial in multimedia applications such as video-conferencing, our collaboration platform exploits locality to efficiently retrieve data. When an application component needs a data segment that is not already replicated locally by other components, a lookup message is broadcasted to the other devices. Every consumer that has this data replies with its delay (in ms) from the original producer. Our collaboration platform obtains the data from the device with lowest latency, when taking network delay and intermediate hops into account. This approach, described in Algorithm 1, finds the path with the lowest possible latency at the given time, but changes in network conditions may require subsequent re-organization of the distribution tree. Re-balancing the distribution tree in such events is left for future work.

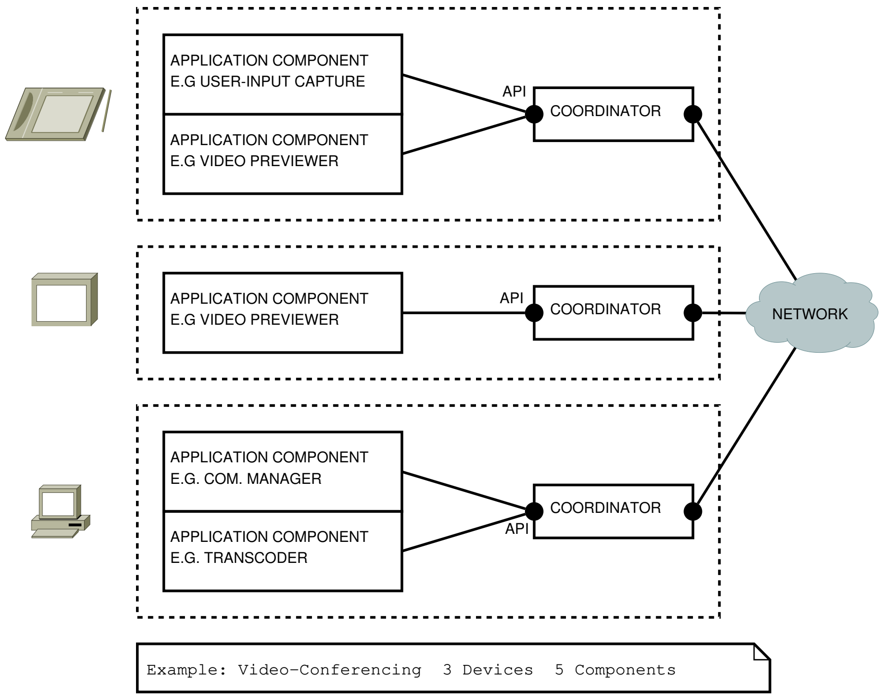


Figure 10.2: Collaboration platform example with three devices and five components working together.

Our collaboration platform consists of two parts, shown in Figure 10.2 and

10.3: A coordinator and the individual application components.

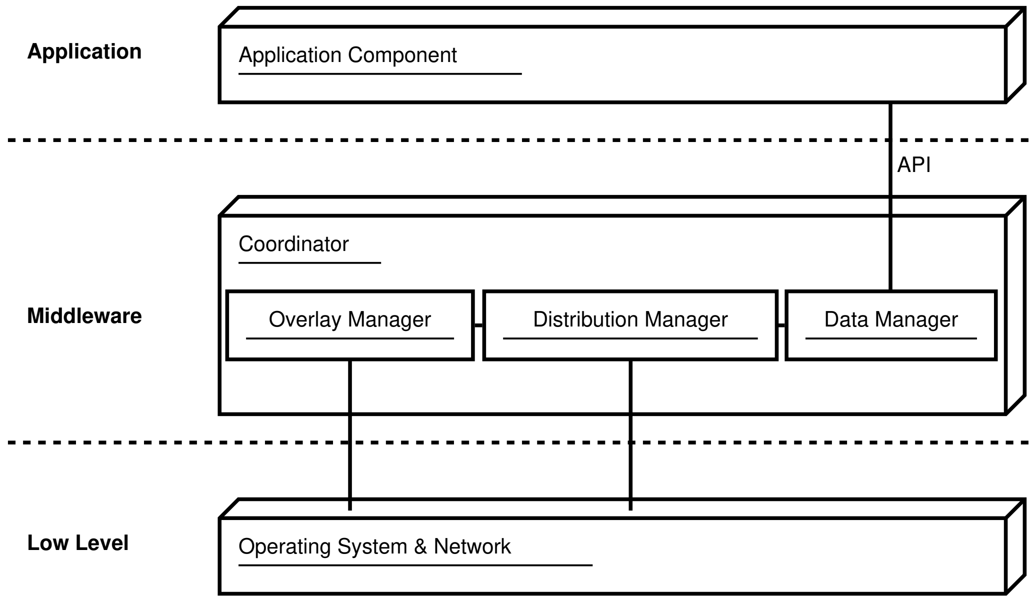


Figure 10.3: System Overview

The coordinator is the main component in our architecture. It is responsible for communication and data management. It has three modules, shown in Figure 10.3. The `overlay manager` is responsible for setting up the network connections in the personal device federation. All devices have connections to all other devices. The `distribution manager` is responsible for creating the distribution trees based on latencies and sending and receiving of data. The `data manager` is responsible for receiving data from application components and delivering data to them when new data segments are produced.

There are two input interfaces to the coordinator, shown as black dots in Figures 10.2 and 10.4. The local component interface is the components' link to the coordinator. The accepted messages form the coordinator API and is used by application developers to share and access data segments between components. The API consists of the following four functions:

A pure producer will use the `Initialize` and `Publish` functions, while a pure consumer will use the `Initialize` and either `Get` or `Subscribe` functions. More complex scenarios where components both produce and consume data segments are possible. The data propagation with `Subscribe` uses a write-driven strategy where producing and replicating devices immediately send data to its subscribers whenever new data becomes available. This is most relevant for

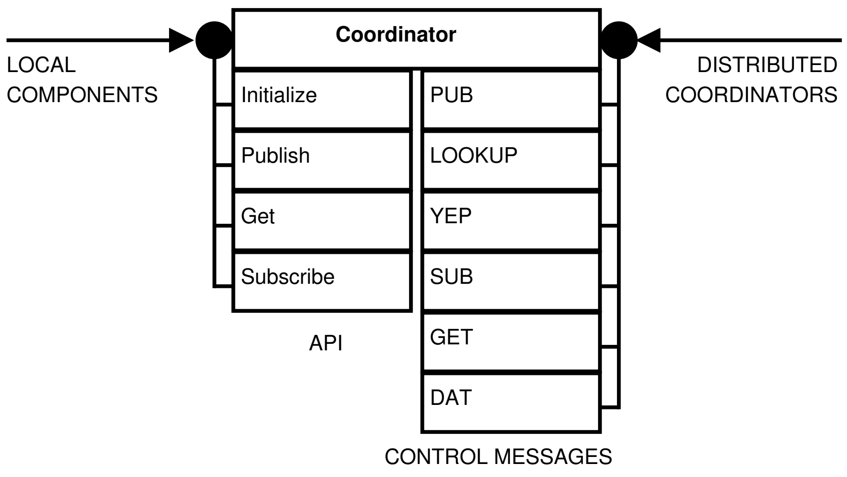


Figure 10.4: Coordinator Interfaces

Function	**Description**
Initialize	Associate a label to a data segment
Publish	Make data segment available to other components
Get	Receive the current instance of the data segment
Subscribe	Receive continuous data segment updates

Table 10.1: Application Component API

multimedia applications. If a producer wants to buffer several data segments, this has to be temporary stored in a different part of memory before being copied to the shared memory structure. `Get` uses a read-driven strategy where the consumer decides when it needs an update of a data segment, relevant for slower paced application domains.

The distributed coordinators' interface is used for underlying control messages between coordinators. These messages are not visible to the application components. The control messages are:

PUB: The receiver is informed that the sender can be used as a source for the data segment identified by the given label.

LOOKUP: The receiver is queried by the sender whether it has the data segment for the given label.

YEP: The receiver registers one possible source for the data segment identified by the given label, and the sender's latency to the original producer.

GET: The sender asks the receiver to send the current instance of the data segment corresponding to the label.

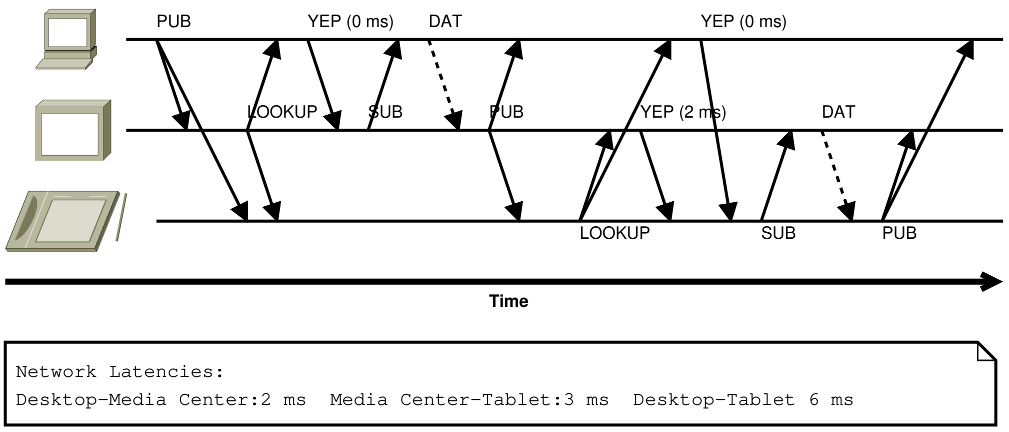
SUB: The sender asks the receiver to send the data segments corresponding to the label continuously.

DAT: The receiver gets a data segment for the given label.

CONTROL MESSAGE	LABEL	SIZE	PAYLOAD
E.G YEP	E.G WEBCAM_BUFFER	E.G 48 KB	E.G 1.641 MS

Figure 10.5: Packet Layout

When a coordinator is receiving many control packets, the packet handling time is increased. This can affect the latency when answering `LOOKUP` messages. Latency can also be affected by underlying network delay. A typical scenario where control messages are sent between coordinators is shown in Figure 10.6.



```
Network Latencies:
Desktop-Media Center:2 ms  Media Center-Tablet:3 ms  Desktop-Tablet 6 ms
```

Figure 10.6: Control traffic example with one producer and two consumers.

In the example scenario in Figure 10.6 we can see how the coordinator sends control messages between the devices, and how Algorithm 1 is used by the Media Center and the Tablet to find their parent in the distribution tree.

In this example the Desktop is producing data and sends a `PUB` message to the two other devices. The Media Center needs this data and broadcasts a `LOOKUP`

Algorithm 1: Latency Optimized Parent Selection

```
Data: A label identifying a data segment.
Result: The data provider with lowest delay for the provided label, or
        NULL if no data providers exist.

$chosen\_parent \longleftarrow NULL$
$optimal\_delay \longleftarrow initial\_threshold$
$start\_time \longleftarrow current\_time$
foreach $peer \in mesh$ do $send\ LOOKUP(label)$;

foreach $received\ reply\ YEP(inherited\_delay)$ do
    $delay \longleftarrow inherited\_delay + (current\_time - start\_time)/2$
    if $delay < optimal\_delay$ then
        $chosen\_parent \longleftarrow peer$
        $optimal\_delay \longleftarrow delay$

    if $current\_time > start\_time + optimal\_delay$ then
        break
```

message. It gets a `YEP` response from the Desktop. The reply specifies 0 ms, meaning that the Desktop is the original producer for this data segment. The Media Center adds the network latency, measured as the time between sending the `LOOKUP` message and receiving the `YEP` message divided by two, to the 0 ms and gets 2 ms latency. The Media Center waits 2 ms for a better offer, but gets no other offers, because the Desktop is the only device with this data segment available. The Media Center sends a `SUB` message to the Desktop indicating that it has chosen it as its parent in the distribution tree for this data segment. The data then flows from the Desktop to the Media Center. A similar sequence of events occurs when the Tablet needs the same data segment, but using Algorithm 1, the Media Center is chosen as its parent. The data flows from the Desktop to the Media Center, and then to the Tablet, providing the lowest latency possible for this data segment to both subscribers.

10.4 Implementation

We have implemented our collaboration platform as a user-space daemon in C. The daemon runs in a stand-alone process and communicates with the application components through a shared library. The shared library is implementing the `Initialize`, `Publish`, `Subscribe` and `Get` functions described in Section

10.3.

The IPC mechanism used between the library and the coordinator is Portable Operating System Interface (POSIX) Shared Memory. It has high throughput and low latency and does not use any copying operations, ensuring that data segments are immediately available to the local components when arriving from the network.

TCP is used as packet transport in order to guarantee that the produced data segments always arrive, that they arrive in order and that they only arrive once at each consumer. We have seen that the implications of using TCP instead of UDP is minimal because all the connections are always open and ready to be used, but the API can be extended with UDP functionality.

10.5 Evaluation

We have two main goals for our evaluation: 1) Verify that our collaboration platform operates according to the design specified in Section 10.3 and that it supports a federation of 15 devices. 2) Evaluate the performance of the platform compared to the requirements of the multimedia application domain. We perform two sets of experiments, one for each of our evaluation goals. To see that our platform works, we look at the sent messages and how the distribution tree is formed. To evaluate the performance of the platform we look at the timestamps of the sent and received data segments.

We have developed a workload generator consisting of a producing and a consuming component sharing a 48 kB data segment, equivalent to a 100 ms buffer of MPEG-2 encoded video. The producer makes 1000 updates to this data segment at two different update rates, resulting in 1 Mbps and 5 Mbps throughput. All updates are timestamped to find the total propagation delay from producer to consumers. Experiments are done twice, sending data from machine A to machine B and again from B to A in order to correct for unsynchronized clocks.

	High Performance HW	**Low Performance HW**
CPU	Intel Core i7, 2.93 GHz	Intel Pentium 4, 1.60 Ghz
RAM	7926 MB	495 MB
NIC	1000 Mbps	100 Mbps
OS	Linux 2.6.18 x86_64	Linux 3.2.0-24 i686

Table 10.2: Hardware and software specifications for the machines used in the experiments.

We use two different sets of machines, labeled *High Performance HW* and *Low Performance HW* in Table 10.2, to find the impact of hardware on our system performance.

In our first set of experiments we observe that our latency optimization and replication technique works correctly. Figure 10.7 shows how three devices share a data segment. *P* is the original producer and *C1*, *C2* and *C3* are consumers. We have federated 15 devices, but leave the remaining 11 out of Figure 10.7 for simplicity.

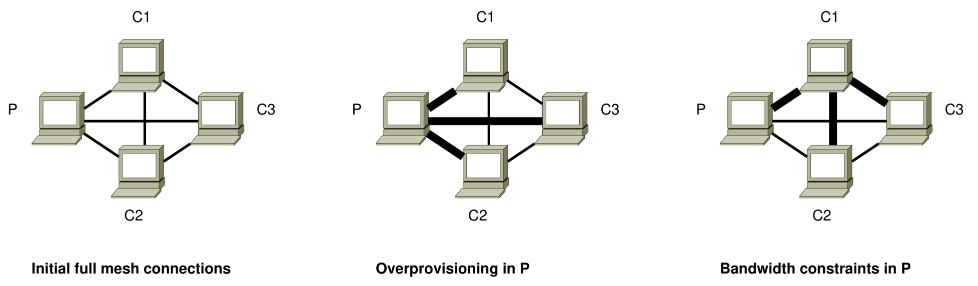


Figure 10.7: Initial experiment setup, data propagation with available bandwidth in P, and data propagation with limited bandwidth in P where C1 is used as replicator.

When *C1*, *C2* and *C3* need a data segment and *P* has available bandwidth, all three consumers receive this data segment directly from *P*. However, when we repeat the experiment with a bandwidth constraint in *P* identical to the throughput of sending data to one consumer, latency is affected and the distribution tree is constructed differently.

		CONSUMERS		
		C1	**C2**	**C3**
	P	1.641 ms	39.919 ms	40.597 ms
REPLICATORS	**C1**		4.263 ms	4.103 ms
	C2			7.147 ms

Table 10.3: Latency vector for data segments as seen from three different consumers, C1 – C3.

Table 10.3 shows the total producer-to-consumer latencies as observed from the consumers when bandwidth in *P* is limited. *C1* detects that it can only get the data segment from *P* with 1.641 ms delay, and it sets *P* as its parent in the distribution tree. At this point *P*'s bandwidth is exhausted. After this, *C2* detects that it can get the data segment from either *P* at 39.919 ms delay or *C1* at 4.263 ms delay. *C2* chooses *C1* to be its parent because it has the lowest delay. The same is true for *C3* which has the option to get the data from either *P* at 40.597 ms, *C1* at 4.103 ms delay or *C2* at 7.147 ms delay. *C3* chooses *C1* to be its parent. This mechanism off-loads *P* and maintains the lowest possible delay for all consumers.

In our second set of experiments we aim to evaluate the performance of our architecture. The application overhead is calculated using $t - n = o$, where t is the total producer-to-consumer delay and n is the network delay.

The network delay is found using the ping tool and dividing the round trip times by 2 to get the time it takes for packets to travel in one direction. 50000 byte packets are used because it is the packet size used in our implementation. Average delay is found using 1000 packets. The delay between *High Performance HW* is $\frac{1.221ms}{2} = 0.6105ms$, and the delay between *Low Performance HW* is $\frac{9.172ms}{2} = 4.586ms$.

	High Perf. HW	**Low Perf. HW**
1 Mbps	1.3572 ms	20.0137 ms
5 Mbps	1.4074 ms	27.6731 ms

Table 10.4: Average producer-to-consumer delay with different throughput on different hardware.

The total producer-to-consumer delay measurements for 1 Mbps and 5 Mbps are shown in Table 10.4, and the application overhead is shown in Table 10.5.

	High Performance HW
1 Mbps	1.3572 ms - 0.6105 ms = 0.7467 ms
5 Mbps	1.4074 ms - 0.6105 ms = 0.7969 ms

	Low Performance HW
1 Mbps	20.0137 ms - 4.586 ms = 15.4277 ms
5 Mbps	27.6731 ms - 4.586 ms = 23.0871 ms

Table 10.5: Application Overhead

We have seen that multi-device multimedia applications are easy to build using our API. Sharing real-time data between different application components is possible because of our latency optimization and replication technique, and delay and jitter can be reduced in multimedia applications. Even with *Low Performance HW* our collaboration platform can distribute data within the time requirements for multimedia applications. Without our system, popular data segments could overload devices and make an entire application unstable. By using our platform, producing components are not bounded by the number of consumers. Instead, consumers provide additional replication and bandwidth.

10.6 Conclusions

We have designed, implemented and tested a collaboration platform for application data sharing, optimized for low producer-to-consumer delay. Distributed applications are easy to build using our API and application components can share data while remaining oblivious to the underlying data propagation. Our experiments show that it is possible to federate 15 devices which is more devices than any of the subjects owned in [53], and more than one person uses concurrently. Distribution trees are built automatically based on latency where the original producer is the root node, replicating consumers are inner nodes and pure consumers are leaf nodes. We have demonstrated that latency increases noticeably when bandwidth is exhausted and that this fact can be exploited to provide load-balancing. The producer-to-consumer delays seen in our experiments are below 100 ms, which is the delay requirement for most multimedia applications.

The collaboration platform that we have presented is part of an ongoing research project for application migration called TRAMP Real-time Application Mobility Platform [187]. The presented system is an important part of the project, and without a system like this, fine-grained application migration is impossible.

Our short-term goal is to re-balance distribution trees when latencies change dramatically. Our long-term goal is to continue the pursuit of ubiquitous multimedia applications by allowing components to migrate between devices and change where they execute based on context.

Chapter 11

P6 – Migration of Fine-grained Multimedia Applications

Authors: Hans Vatne Hansen, Francisco Javier Velázquez-García, Vera Goebel, Ellen Munthe-Kaas, Thomas Plagemann

Published in: Proceedings of the Posters and Demo Track, Middleware '12

Reference in Bibliography: [96]

Abstract: In order to leverage the potential of the device diversity of users, we aim to provide a middleware solution where parts of a multimedia application migrate to different devices and take advantage of more processing power and different I/O capabilities. The middleware is fully designed, and partially implemented and evaluated. Preliminary results from location transparent data distribution and seamless connection handover are promising with respect to throughput and latency requirements for multimedia applications.

11.1 Introduction

People own an increasing number of multimedia capable devices, such as smart phones, laptops and media centers. These devices differ with respect to mobility, processing power, and I/O capabilities. Current research in the area of multi-device applications aims to reduce the needed efforts to adapt applications to different I/O capabilities, to allow users to run these applications on most or all of their devices. While this is an important step towards user-friendliness, one inherent limitation that we aim to overcome is that multi-device applications are designed to run on a single device at a time.

We aim to enable future applications to dynamically utilize the devices in the vicinity for different tasks by running parts of the application on different devices. This is especially beneficial for multimedia applications to leverage the characteristics of different devices for the different media types. For example, an application can display the video on the largest available screen and play the audio on a hi-fi equipped media center, and still function as one application. Therefore, it is necessary to additionally support migration of application parts, i.e., fine-grained migration. In order to do this, we retake process migration research in the context of the new (mobile) devices and multimedia requirements. In particular, we are developing the TRAMP Real-time Application Mobility Platform which aims to support fine-grained migration between heterogeneous devices with low freeze time to improve Quality of Experience and minimize administration efforts (i.e., install and configure once, and use on all devices). The three core ideas of TRAMP are:

(1) Efficient data sharing between the application parts, both if they run on the same device, and on different devices. The idea is to build distribution trees based on latency where all consumers of data are potential replicators. Since the location of execution must be transparent to the application developer, a common abstraction and API for local and remote data sharing is provided by TRAMP.

(2) TRAMP provides mobility transparency in IP networks with a connection handover system. This system performs socket migration based on proxy forwarding. It enables applications in migration scenarios to interact with legacy applications, such as Spotify and Skype.

(3) We leverage the fact that the devices belong to a single user by creating federations of trusted devices, called device communities. These communities minimize the configuration complexity for users and provide authentication, authorization and data transfer in communities. A device can belong to several communities, such as personal-devices and work-devices.

Migration has been thoroughly researched in the past [140], however, to the best of our knowledge, no migration system for fine-grained multimedia applications exists. Existing migration systems fail to fulfill all requirements of fine-grained multimedia applications, such as low freeze time, support for heterogeneous devices, support for IP mobility, and management of security and trust.

Our vision of pervasive computing is similar to Mobile Gaia [171], but Mobile

Gaia does not provide migration and relies on designated and manually configured coordinators. CloneCloud [45] and MAUI [50] are fine-grained migration systems that off-load processing from mobile devices to improve processing performance and conserve battery. However, they do not address multimedia applications requirements and their statically configured servers do not allow these applications to benefit from the different I/O capabilities of users' devices.

11.2 Design

Our design is based on insights from the state-of-the-art of fine-grained applications and previous process migration research. TRAMP targets multimedia applications and it uses processes as units of execution. Since migration can occur at any time, developers must be agnostic of when and where processes run. To minimize this effort, TRAMP provides location transparent communication mechanisms.

We have designed the TRAMP architecture with components that (1) create device communities, (2) migrate processes, (3) perform signalling for process migration, (4) achieve transparent connection handover, (5) provide efficient data sharing, (6) describe and discover application components within device communities, (7) implement policies, and (8) aid users in configuring devices.

(1) The community component creates trusted federations of devices. A well-connected mesh topology is suitable for personal device communities, while DHTs scales better for larger communities. Certificates and pre-shared keys can provide authentication, authorization and encryption.

(2) The process migrator component provides functions to export and import the static and dynamic state in the source node and destination node respectively. The migrator is responsible for sending this data, killing, and resuming the migrated process. To support heterogeneous devices, an abstraction layer to hide different architectures is provided. We propose to use virtual machines, because multimedia applications perform well on heterogeneous mobile devices with current virtual machines, such as Android's Dalvik VM.

(3) The signalling component provides offer/answer mechanisms for process migration, and negotiates requirements of the migrating component. The signaling protocol can be realized by reusing the Session Initiation Protocol (SIP) with process migration semantics.

(4) The connection handover component called SOCKMAN ensures that con-

nections from the endpoint of a fine-grained application to legacy applications are preserved after migration [9]. The solution uses a proxy to tunnel entire IP packets between itself and the migrating component using UDP, preserving the end-to-end principle.

(5) To achieve efficient data sharing, a Distributed Shared Memory (DSM) component enables low-latency communication [95]. It provides communication between local components at speeds equivalent to regular memory operations. In distributed environments, the data is obtained from replicating devices using a latency-optimized distribution tree. Components can communicate with all other components using a location transparent, label-based lookup. This location transparency allows application developers to focus on the task of the component, rather than implementing distribution and coordination techniques.

(6) The description and discovery component advertises and registers components that are available in device communities. It can apply existing protocols like the Simple Service Discovery Protocol (draft-cai-ssdp-v1-03) or the Service Location Protocol (RFC 2608).

(7) The policy component contains user preferences and enables autonomous migration by invoking the migrator component, for example to automatically move a component when a certain device is in the user's vicinity. Another use of policies is to control the availability of components to different migration communities.

(8) The GUI component aids users to configure their devices, to create or join communities and migrate their applications.

We envision a minimal required set of components to be pre-installed in users' devices. These are the community component, the migrator and the signaler. All other TRAMP components can migrate on demand. This simplifies upgrading, because users only need to update a component on one device and it migrates to all the other devices.

11.3 Status and Challenges

From the presented architecture, we have designed, implemented and evaluated the DSM component and the connection handover component. The remaining components are under development.

In [95] we show that our DSM system works and that it is able to provide data sharing throughput of 1 – 5 Mbps with less than 30 ms latency. The evaluation of

SOCKMAN [9] shows that the average socket migration time is 0.218 ms, and that the application is able to reach 1 Gbps of throughput. These results indicate that the evaluated components are suitable for fine-grained multimedia applications.

Open research questions include how to adapt applications to different devices with respect to e.g. screen size, without involving the application developer. We aim to reuse insights from multi-device research such as [114]. Another open question is whether thread migration is better than process migration and if it is possible to make a virtual machine that runs only threads.

Part III

Appendix

Appendix A

Errata

In Figures 6.2 and 7.3 there is an error on the description of Path w_7. Path w_7 should be represented by two paths: w_7, w_8. Figure A.1 is the correction of Figure 6.2, which shows the graph abstraction of the multimedia pipeline in one peer of a video conferencing application before and after adaptation. Figure A.2 is the correction of Figure 7.3. The caption of Figure A.2 is also improved.

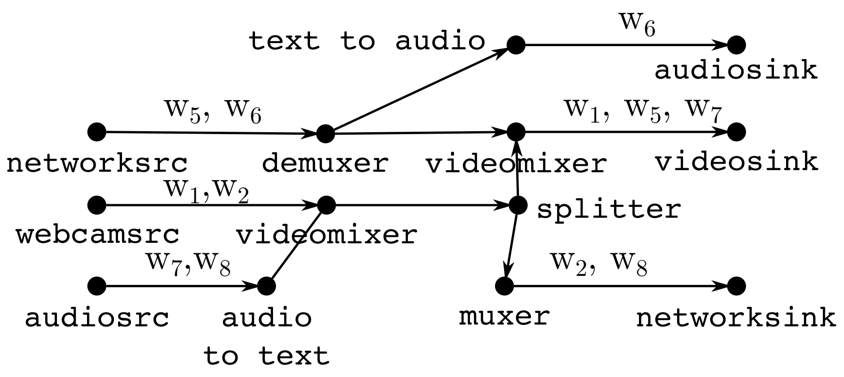


Figure A.1: Errata of Figure 6.2. Path w_7 in Figure 6.2 is detailed as w_7, w_8 in this figure.

The mistake in Figure 6.2 is reflected in the text in Section 6.2.1, which reads:

> In Figure 6.2, Path w_4 has adapted to Path w_6, and Path w_3 has adapted to Path w_7. Paths w_6, and w_7 convert text modality to audio, and audio to text respectively.

The correct text should read:

In Figure 6.2, Path w_4 has adapted to Path w_6, and Path w_3 has adapted to Paths w_7 and w_8. Path w_6, converts *text to audio*, and Paths w_7, w_8 convert *audio to text*. Path w_7 renders the converted text to the display, and Path w_8 sends the converted text over the network.

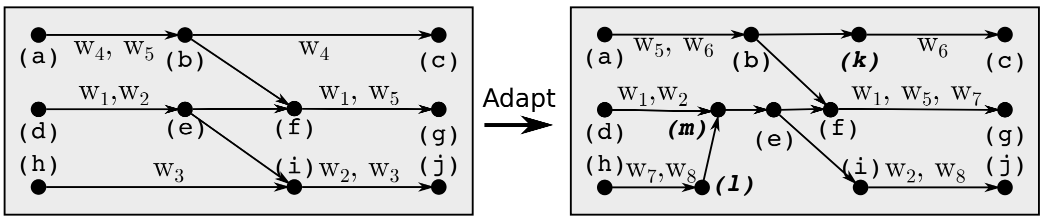


Figure A.2: Graph abstraction of the multimedia pipeline in peer *A* of a video conferencing application before and after adaptation. On the left, the pipeline consumes and produces video and audio in peer *A*. Then, peer *B* for some reason cannot process audio anymore, but can continue processing video and text. The user of peer *A* prefers to interact with audio and video modalities, but no text, and his host device contains components to convert *audio to text* (from his microphone) and *text to audio* (from the data received from peer *B*). Therefore, peer *A* adapts its pipeline (as shown on the right side of the figure) in a way that it converts the modalities in components `(l)` and `(k)`. As a result, the user at peer *A* continues the interaction with the application with audio and video modalities, while the user at peer *B* interacts with video and text modalities. The vertices in the figure represent the following components: (a) `networksrc`, (b) `demuxer`, (c) `audiosink`, (d) `webcamsrc`, (e) `splitter`, (f) `videomixer`, (g) `videosink`, (h) `audiosrc`, (i) `muxer`, (j) `networksink`, (k) `text-to-audio`, (l) `audio-to-text`, and (m) `text-overlay`. $\{w\}_1^8$ represent functional paths. Note that path w_4 is adapted into path w_6, and path w_3 is adapted into paths w_7, w_8.

Appendix B

Additional Use Cases

In this appendix, we describe a series of use cases (hypothetical scenarios) that exemplify the goals (in Section 1.3), and requirements (in Section 2.3) of this thesis. We foresee the behavior of the applications in these use cases as if they were developed with the proposed Application Program Interfaces (APIs) in this thesis.

B.1 Augmented Reality

Bob suffers of Alzheimer, therefore he carries a wearable computer with eyeglasses that display the name of the people he encounters. Since the battery and heat dissipation of the eyeglasses is not optimal for CPU intensive tasks such a face recognition, the eyeglasses have only installed a light version of the middleware (as specified in Section 11.2), a pipeline for capturing video from a built-in camera, and a pipeline for rendering the overlays of recognized people. The definition of the pipeline to perform face recognition is in the eye glasses; this pipeline is to be moved and instantiated in a device in the vicinity as the patient walks into different environments.

Similar usages of mobile pipelines for eyeglasses with augmented reality can be useful for people suffering of autism. With the offloading of pipelines that use artificial intelligence algorithms, the eyeglasses can display hints on how to behave in a certain situation. The eyeglasses here envisioned will be an evolution from current products such as Glass from Google[1] or HoloLens 2 from Microsoft[2].

[1]https://developers.google.com/glass/

[2]https://www.microsoft.com/en-us/hololens/

B.2 Travel Assistance

Alice is planning her journey using the travel assistance application as in [93]. She started the application on her mobile device while descending to the station on the escalator. When she passes a kiosk computer on the train platform, she accepts the suggestion from the application to take advantage of the kiosk's larger and easier to use display instead. Alice moves the video modality to the kiosk. When Alice walks away from the kiosk, the video modality is back to her mobile device.

B.3 Mobile Application between Fixed Devices

Bob is doing some edition in a program, e.g. coding, movie editing, or graphic design. Then, he is asked to present his work to an audience. Bob considers the audience will understand his work much better if he gives a demo on how he is doing the actual work. He does not have his laptop at hand, so he moves the application from the desktop computer in his office to the desktop computer (with an attached projector) in the meeting room.

B.4 Video Conferencing at Home

Alice starts a video conferencing session with her mother on her desktop computer at home. When Alice's husband and children arrive at home, Alice moves the application to the living room in the following manner. The video processing is moved to the TV, the camera capture is moved to a dedicated wireless webcam, the audio reproduction is moved to the living room audio system, the audio capture is moved to the microphone in the wireless webcam, and the Graphical User Interface (GUI) of application control is moved to her mobile phone.

B.5 Video Conferencing in Transit and Modality Change

Alice starts a video conferencing session with Bob when she is at home. The initial configuration of the multimedia session makes Alice and Bob producers and consumers of video and audio modalities. At home, Alice uses her desktop computer. Bob communicates from a remote office with a dedicated video conferencing system.

Alice has to travel to her office by train, so she moves the application to her mobile phone. When she enters in a wagon where speaking by telephone is forbidden, the application changes the audio modality to text modality. That is, the application converts the audio from Bob into text, and the text input (from a virtual keyboard on display) from Alice is converted to audio at Bob's device. When Alice leaves the wagon, the application resumes the processing of audio modality.

Bibliography

[1] Mourad Alia et al. “A Component-Based Planning Framework for Adaptive Systems.” In: *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*. Berlin, Heidelberg: Springer Berlin Heidelberg, Oct. 2006, pp. 1686–1704.

[2] Mourad Alia et al. “A Utility-Based Adaptivity Model for Mobile Applications.” In: *Proc. of AINAW* (2007). DOI: 10.1109/ainaw.2007.64.

[3] Mourad Alia et al. “Managing Distributed Adaptation of Mobile Applications.” In: *Distributed Applications and Interoperable Systems*. Springer Berlin Heidelberg, June 2007, pp. 104–118. ISBN: 978-3-540-72881-8. DOI: 10.1007/978-3-540-72883-2_8.

[4] Mark Allman and Aaron Falk. “On the Effective Evaluation of TCP.” In: *SIGCOMM Comput. Commun. Rev.* 29.5 (1999), pp. 59–70. DOI: 10.1145/505696.505703.

[5] B Alpern et al. “The Jikes Research Virtual Machine project: Building an open-source research community.” In: *IBM Systems Journal* 44.2 (2005). DOI: 10.1147/sj.442.0399.

[6] Sten L Amundsen and Frank Eliassen. “A resource and context model for mobile middleware.” In: *Personal and Ubiquitous Computing* 12.2 (Oct. 2006).

[7] Sten Lundesgaard Amundsen and Frank Eliassen. “Combined Resource and Context Model for QoS-Aware Mobile Middleware.” In: *Architecture of Computing Systems - ARCS 2006*. Berlin, Heidelberg: Springer Berlin Heidelberg, Mar. 2006, pp. 84–98. ISBN: 978-3-540-32765-3. DOI: 10.1007/11682127_7.

[8] S Amundsen et al. “QuA: platform-managed QoS for component architectures.” In: *Proceedings of Norwegian Informatics Conference (NIK)*. 2004.

[9] Håvard Stigen Andersen. “User Space Socket Migration for Mobile Applications.” [Online `http://urn.nb.no/URN:NBN:no-32922`; accessed: 2018-09-09. MA thesis. Universitetet i Oslo, May 2012.

[10] Marko Andic. “Negotiation and Data Transfer for Application Mobility.” [Online `http://urn.nb.no/URN:NBN:no-48141`; accessed: 2018-09-09]. MA thesis. University of Oslo, 2015.

[11] Z Anwar et al. “Plethora: a framework for converting generic applications to run in a ubiquitous environment.” In: *Mobile and Ubiquitous Systems: Networking and Services, 2005. MobiQuitous 2005. The Second Annual International Conference on*. 2005. DOI: `10.1109/MOBIQUITOUS.2005.47`.

[12] Giorgio Ausiello et al. *Complexity and Approximation*. Combinatorial Optomization Problems and Their Approximability Properties. Springer Berlin Heidelberg, 1999. ISBN: 978-3-642-63581-6. DOI: `10.1007/978-3-642-58412-1`.

[13] Rajesh Krishna Balan and Jason Flinn. “Cyber Foraging: Fifteen Years Later.” In: *IEEE Pervasive Computing* 16.3 (2017), pp. 24–30. DOI: `10.1109/mprv.2017.2940972`.

[14] Rajesh Krishna Balan et al. “Simplifying cyber foraging for mobile devices.” In: *MobiSys ’07: Proceedings of the 5th international conference on Mobile systems, applications and services*. ACM Request Permissions, June 2007. DOI: `10.1145/1247660.1247692`.

[15] Rajesh Balan et al. “The Case for Cyber Foraging.” In: *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. EW 10. Saint-Emilion, France: ACM, 2002, pp. 87–92. DOI: `10.1145/1133373.1133390`.

[16] M Baldi and Y Ofek. “End-to-end delay analysis of videoconferencing over packet-switched networks.” In: *IEEE/ACM Transactions on Networking* 8.4 (2000), pp. 479–492. DOI: `10.1109/90.865076`.

[17] Guruduth Banavar et al. “Challenges: an application model for pervasive computing.” In: *Proceedings of the 6th annual international conference on Mobile computing and networking - MobiCom ’00* (2000). DOI: 10.1145/345910.345957.

[18] M Bashari, E Bagheri, and W Du. “Dynamic software product line engineering: a reference framework.” In: (Feb. 2016).

[19] Mahdi Bashari, Ebrahim Bagheri, and Weichang Du. “Dynamic Software Product Line Engineering: A Reference Framework.” In: *International Journal of Software Engineering and Knowledge Engineering* 27.2 (2017), pp. 191–234. DOI: 10.1142/S0218194017500085.

[20] P van Beek et al. “Metadata-driven multimedia access.” In: *Signal Processing Magazine, IEEE* 20.2 (2003), pp. 40–52. DOI: 10.1109/MSP.2003.1184338.

[21] Tom Beigbeder et al. “The effects of loss and latency on user performance in unreal tournament 2003 #174.” In: *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. ACM, 2004, pp. 144–151. ISBN: 1-58113-942-X. DOI: 10.1145/1016540.1016556.

[22] Nelly Bencomo et al. *Models@run.time*. Foundations, Applications, and Roadmaps. Springer, July 2014. ISBN: 3319089153.

[23] John K. Bennett, John B. Carter, and Willy Zwaenepoel. “Munin: Distributed Shared Memory Based on Type-specific Memory Coherence.” In: *SIGPLAN Not.* 25.3 (Feb. 1990), pp. 168–176. ISSN: 0362-1340. DOI: 10.1145/99164.99182.

[24] Massimo Bernaschi, Francesco Casadei, and Paolo Tassotti. “SockMi: A solution for migrating TCP/IP connections.” In: *Parallel, Distributed and Network-Based Processing (PDP), 15th EUROMICRO International Conference on*. 2007, pp. 221–228.

[25] Krishna A. Bharat and Luca Cardelli. “Migratory Applications.” In: *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*. UIST ’95. Pittsburgh, Pennsylvania, USA: ACM, 1995, pp. 132–142. ISBN: 0-89791-709-X. DOI: 10.1145/215585.215711.

[26] Krishna Bharat and Marc H. Brown. “Building distributed, multi-user applications by direct manipulation.” In: *Proceedings of the 7th annual ACM symposium on User interface software and technology - UIST ’94* (1994). DOI: `10.1145/192426.192454`.

[27] Andrew P Black et al. “Infopipes: An abstraction for multimedia streaming.” In: *Multimedia Systems* 8.5 (2002), pp. 406–419. DOI: `10.1007/s005300200062`.

[28] Gordon Blair and Paul Grace. “Emergent Middleware: Tackling the Interoperability Problem.” In: *Ieee Internet Computing* 16.1 (2012), pp. 78–81. DOI: `10.1109/MIC.2012.7`.

[29] R Bless et al. “The Underlay Abstraction in the Spontaneous Virtual Networks (SpoVNet) Architecture.” In: *Next Generation Internet Networks, 2008. NGI 2008* (2008), pp. 115–122. DOI: `10.1109/NGI.2008.22`.

[30] M Bonola and S Salsano. “Per-application Mobility management: Performance evaluation of the UPMT solution.” In: *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*. July 2011, pp. 2249–2255. DOI: `10.1109/IWCMC.2011.5982892`.

[31] Marco Bonola, Stefano Salsano, and Andrea Polidoro. “UPMT: Universal Per-application Mobility management using Tunnels.” In: *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*. IEEE, 2009, pp. 1–8.

[32] Stefan Bosse. “VAMNET: the functional approach to distributed programming.” In: *SIGOPS Operating Systems Review* 40.3 (July 2006). DOI: `10.1145/1151374.1151376`.

[33] G Bouabene, C Jelger, and C Tschudin. “The Autonomic Network Architecture (ANA).” In: *Selected Areas in Communications, IEEE Journal on* (2010).

[34] Terrehon Bowden et al. *The /proc filesystem*. `https://www.kernel.org/doc/Documentation/filesystems/proc.txt`. [Online; accessed: 2018-06-30]. June 2009.

[35] G Brataas et al. “Scalability of Decision Models for Dynamic Product Lines.” In: *SPLC* (2007).

[36] Mark Burgess and L Kristiansen. “On the complexity of determining autonomic policy constrained behaviour.” In: *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE* (2008), pp. 295–301. DOI: 10.1109/NOMS.2008.4575147.

[37] J Bush, J Irvine, and J Dunlop. “Removing The Barriers to Ubiquitous Services: A User Perspective.” In: *Mobile and Ubiquitous Systems - Workshops, 2006. 3rd Annual International Conference on*. IEEE, 2006, pp. 1–5. ISBN: 0-7803-9791-6. DOI: 10.1109/MOBIQW.2006.361746.

[38] Giacomo Cabri, Letizia Leonardi, and Raffaele Quitadamo. “Enabling Java mobile computing on the IBM Jikes research virtual machine.” In: *Proceedings of the 4th international symposium on Principles and practice of programming in Java*. ACM, 2006, pp. 62–71. ISBN: 3-939352-05-5. DOI: 10.1145/1168054.1168064.

[39] Julio Cano, Natividad Martinez Madrid, and Ralf Seepold. “OSGi services design process using model driven architecture.” In: *2009 IEEE/ACS International Conference on Computer Systems and Applications* (2009). DOI: 10.1109/aiccsa.2009.5069418.

[40] L Cardelli. “A language with distributed scope.” In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT*. 1995.

[41] K Carey, K Feeney, and D Lewis. “State of the Art: Policy Techniques for Adaptive Management of Smart Spaces.” In: *State of the Art Surveys* (2003).

[42] John B. Carter, John K. Bennett, and Willy Zwaenepoel. “Implementation and Performance of Munin.” In: *SIGOPS Oper. Syst. Rev.* 25.5 (Sept. 1991), pp. 152–164. ISSN: 0163-5980. DOI: 10.1145/121133.121159.

[43] Soraya Ait Chellouche et al. “Context-aware multimedia services provisioning in future Internet using ontology and rules.” In: *Network of the Future (NOF), 2014 International Conference and Workshop on the* (2014), pp. 1–5. DOI: 10.1109/NOF.2014.7119778.

[44] Shang-Wen Cheng. “Rainbow: Cost-effective Software Architecture-based Self-adaptation.” PhD thesis. Pittsburgh, PA, USA: Carnegie Mellon University, 2008. ISBN: 978-0-549-52525-7.

[45] Byung-Gon Chun et al. “CloneCloud: Elastic Execution between Mobile Device and Cloud.” In: *the sixth conference*. New York, New York, USA: ACM Press, 2011, p. 301. ISBN: 9781450306348. DOI: `10.1145/1966445.1966473`.

[46] X.Org Community and X.Or Foundation. *X Window System*. `https://www.x.org/releases/X11R7.7/`. Version 11. [Online; accessed: 2018-08-12]. June 2012.

[47] Antonio Corradi, Franco Zambonelli, and Letizia Leonardi. “A Scalable Tuple Space Model for Structured Parallel Programming.” In: *Proceedings of the Conference on Programming Models for Massively Parallel Computers*. PMMP ’95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 25–32. ISBN: 0-8186-7177-7.

[48] Cristiano Andre da Costa, Adenauer Correa Yamin, and Claudio Fernando Resin Geyer. “Toward a General Software Infrastructure for Ubiquitous Computing.” In: *IEEE Pervasive Computing* 7.1 (Jan. 2008), pp. 64–73. ISSN: 1536-1268. DOI: `10.1109/mprv.2008.21`.

[49] P. Couderc and M. Banatre. “Ambient computing applications: an experience with the SPREAD approach.” In: *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*. Jan. 2003. DOI: `10.1109/HICSS.2003.1174830`.

[50] Eduardo Cuervo et al. “MAUI: making smartphones last longer with code offload.” In: *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62. ISBN: 978-1-60558-985-5. DOI: `10.1145/1814433.1814441`.

[51] Garlan David. *ABLE Research Group – Channging Architecture*. `https://www.cs.cmu.edu/~able/index.html`. [Online; accessed: 2018-09-23]. 2018.

[52] Katrien De Moor et al. “Proposed Framework for Evaluating Quality of Experience in a Mobile, Testbed-oriented Living Lab Setting.” In: *Mobile Networks and Applications* 15.3 (2010), pp. 378–391. DOI: `10.1007/s11036-010-0223-0`.

[53] David Dearman and Jeffery S Pierce. “It’s on my other computer!: computing with multiple devices.” In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM,

2008, pp. 767–776. ISBN: 978-1-60558-011-1. DOI: 10.1145/1357054.1357177.

[54] P.J. Denning et al. “Computing as a discipline.” In: *Computer* 22.2 (Feb. 1989), pp. 63–70. ISSN: 0018-9162. DOI: 10.1109/2.19833.

[55] A K Dey. “Providing architectural support for building context-aware applications.” PhD thesis. Georgia Institute of Technology, 2000.

[56] Anind K Dey. “Understanding and Using Context.” In: *Personal and Ubiquitous Computing* 5.1 (Jan. 2001), pp. 4–7. DOI: 10.1007/s007790170019.

[57] Simon DiMaio, Mike Hanuschik, and Usha Kreaden. “The da Vinci Surgical System.” In: *Surgical Robotics* (Nov. 2010), pp. 199–217. DOI: 10.1007/978-1-4419-1126-1_9.

[58] Dolphin Interconnect Solutions. *SuperSockets for Linux Overview*. https://www.dolphinics.com/download/WHITEPAPERS/Dolphin_Express_IX_SuperSockets_for_Linux.pdf. [Online; accessed: 2018-09-03]. Aug. 2013.

[59] Fred Douglis et al. “A Comparison of Two Distributed Systems: Amoeba and Sprite.” In: *Computing Systems*. 1991.

[60] J Echaiz and J R Ardenghi. “Security in process migration systems.” In: *Journal of Computer Science & Technology* (2005). Ed. by Ramiro Jordán and Fernando Tinetti. ISSN: 1666-6038.

[61] Viktor S Wold Eide et al. “Fine granularity adaptive multireceiver video streaming.” In: *Multimedia Computing and Networking 2007* 6504 (Jan. 2007), 65040O–65040O–11. DOI: 10.1117/12.709775.

[62] Heinz-Josef Eikerling and Frank Berger. “Design of OSGi Compatible Middleware Components for Mobile Multimedia Applications.” In: *Protocols and Systems for Interactive Distributed Multimedia*. Springer Berlin Heidelberg, Nov. 2002, pp. 80–91. ISBN: 978-3-540-00169-0. DOI: 10.1007/3-540-36166-9_8.

[63] Frank Eliassen et al. “Evolving self-adaptive services using planning-based reflective middleware.” In: *ARM@Middleware* (2006), p. 1. DOI: 10.1145/1175855.1175856.

[64] Ernesto Exposito and Jorge Gómez-Montalvo. “An Ontology-Based Framework for Autonomous QoS Management in Home Networks.” In: *2010 Sixth International Conference on Networking and Services (ICNS)*. IEEE, 2010, pp. 117–122. ISBN: 978-1-4244-5927-8. DOI: 10.1109/ICNS.2010.24.

[65] L. L. Fernández et al. “Kurento: a media server technology for convergent WWW/mobile real-time multimedia communications supporting WebRTC.” In: *Proc. of WoWMoM*. 2013, pp. 1–6. DOI: 10.1109/WoWMoM.2013.6583507.

[66] S J Fink, Feng Qian Code Generation, and 2003 CGO 2003 International Symposium on Optimization. “Design, implementation and evaluation of adaptive recompilation with on-stack replacement.” In: *Code Generation and Optimization, 2003. CGO 2003. International Symposium on* (2003), pp. 241–252. DOI: 10.1109/CGO.2003.1191549.

[67] J Floch et al. “Using architecture models for runtime adaptability.” In: *IEEE Software* 23.2 (2006). DOI: 10.1109/MS.2006.61.

[68] *Free and Open Source Software Development European Meeting (FOSDEM)*. [Online https://archive.fosdem.org/2018/; accessed: 2018-08-12]. Brussels, Belgium, 2018.

[69] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. 1st. Addison-Wesley Longman Ltd., 1999.

[70] A. Fuggetta, G.P. Picco, and G. Vigna. “Understanding code mobility.” In: *IEEE Transactions on Software Engineering* 24.5 (May 1998), pp. 342–361. ISSN: 0098-5589. DOI: 10.1109/32.685258.

[71] D Funato, K Yasuda, and H Tokuda. “TCP-R: TCP Mobility Support for Continuous Operation.” In: *1997 International Conference on Network Protocols*. IEEE Comput. Soc, 1997, pp. 229–236. ISBN: 0-8186-8061-X. DOI: 10.1109/ICNP.1997.643720.

[72] D. Garlan et al. “Project Aura: toward distraction-free pervasive computing.” In: *IEEE Pervasive Computing* 1.2 (Apr. 2002), pp. 22–31. ISSN: 1536-1268. DOI: 10.1109/mprv.2002.1012334.

[73] D. Garlan et al. “Rainbow: Architecture-Based Self-Adaptation With Reusable Infrastructure.” In: *Computer* 37.10 (2004), pp. 46–54. DOI: 10.1109/mc.2004.175.

[74] K Geihs et al. “A comprehensive solution for application-level adaptation.” In: *Software: Practice and Experience* 39.4 (Mar. 2009), pp. 385–422. DOI: 10.1002/spe.900.

[75] David Gelernter. “Generative communication in Linda.” In: *ACM Trans. Program. Lang. Syst.* 7.1 (Jan. 1985), pp. 80–112. ISSN: 0164-0925. DOI: 10.1145/2363.2433.

[76] Vivian Genaro Motti. *A computational framework for multi-dimensional context-aware adaptation*. ACM, June 2011. ISBN: 978-1-4503-0670-6. DOI: 10.1145/1996461.1996545.

[77] Simon Giesecke, Wilhelm Hasselbring, and Matthias Riebisch. “Classifying architectural constraints as a basis for software quality assessment.” In: *Advanced Engineering Informatics* 21.2 (2007), pp. 169–179. ISSN: 1474-0346. DOI: 10.1016/j.aei.2006.11.002.

[78] Alejandro Martín Medrano Gil et al. “Separating the Content from the Presentation in AAL: The universAAL UI Framework and the Swing UI Handler.” In: *Advances in Intelligent Systems and Computing* (2013), pp. 113–120. ISSN: 2194-5365. DOI: 10.1007/978-3-319-00566-9_15.

[79] Eli Gjørven et al. “Self-adaptive systems: a middleware managed approach.” In: *SelfMan’06: Proceedings of the Second IEEE international conference on Self-Managed Networks, Systems, and Services*. Springer-Verlag, June 2006, pp. 15–27. ISBN: 978-3-540-34739-2. DOI: 10.1007/11767886_2.

[80] Robert L. Glass. “A structure-based critique of contemporary computing research.” In: *Journal of Systems and Software* 28.1 (Jan. 1995), pp. 3–7. ISSN: 0164-1212. DOI: 10.1016/0164-1212(94)00077-z.

[81] Mark S. Gordon et al. “COMET: Code Offload by Migrating Execution Transparently.” In: *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. 2012, pp. 93–106.

[82] Josh Graessley, Tommy Pauly, and Eric Kinnear. “Introducing Network.framework: A modern alternative to Sockets.” In: *Apple Worldwide Developers Conference (WWDC)*. 2018.

[83] Tomas Gryczon. “Component-based multimedia application for fine-grained migration.” Norwegian. [Online `http://urn.nb.no/URN:NBN:no-44699`; accessed: 2018-09-09]. MA thesis. University of Oslo, 2014.

[84] *GStreamer Autumn Hackfest*. [Online `https://wiki.gnome.org/Hackfests/GstAutumnHackfest2016`; accessed: 2018-08-12]. Berlin, Germany: GNOME, Oct. 2016.

[85] *GStreamer Autumn Hackfest*. [Online `https://wiki.gnome.org/Hackfests/GstAutumnHackfest2017`; accessed: 2018-08-12]. Prague, Czech Republic: GNOME, Oct. 2017.

[86] GStreamer community. *GStreamer Open Source Multimedia Framework*. `https://gstreamer.freedesktop.org/`. [Online; accessed: 2018-03-29].

[87] GStreamer Open Source Multimedia Framework. *GStreamer applications*. 2018.

[88] *GStreamer Spring Hackfest*. [Online `https://wiki.gnome.org/Hackfests/GstHackfest2015`; accessed: 2018-08-12]. Staines, United Kingdom: GNOME, Mar. 2015.

[89] *GStreamer Spring Hackfest*. [Online `https://wiki.gnome.org/Hackfests/GstSpringHackfest2016`; accessed: 2018-08-12]. Thessaloniki, Greece: GNOME, May 2016.

[90] *GStreamer Spring Hackfest*. [Online `https://wiki.gnome.org/Hackfests/GstSpringHackfest2017`; accessed: 2018-08-12]. A Coruña, Spain: GNOME, May 2017.

[91] *GStreamer Spring Hackfest*. [Online `https://wiki.gnome.org/Hackfests/GstSpringHackfest2018`; accessed: 2018-08-12]. Lund, Sweden: GNOME, May 2018.

[92] Tao Gu, H K Pung, and Da Qing Zhang. “A middleware for building context-aware mobile services.” In: *Vehicular Technology Conference, 2004. VTC 2004-Spring. 2004 IEEE 59th*. 2004, pp. 2656–2660. DOI: `10.1109/VETECS.2004.1391402`.

[93] S. Hallsteinsen et al. “A development framework and methodology for self-adapting applications in ubiquitous computing environments.” In: *Journal of Systems and Software* 85.12 (2012), pp. 2840–2859. ISSN: 0164-1212. DOI: 10.1016/j.jss.2012.07.052.

[94] Hans Vatne Hansen, Vera Goebel, and Thomas Plagemann. “DevCom: Device communities for user-friendly and trustworthy communication, sharing, and collaboration.” In: *Computer Communications* 85 (July 2016), pp. 14–27. DOI: 10.1016/j.comcom.2016.02.001.

[95] Hans Vatne Hansen et al. “Efficient Data Sharing for Multi-device Multimedia Applications.” In: *Proceedings of the Workshop on Multi-device App Middleware*. Ed. by Christian Fuhrhop, Stephan Steglich, and Ajit Jaokar. Multi-Device ’12. Montreal, Quebec, Canada: ACM, 2012, 2:1–2:6. ISBN: 978-1-4503-1617-0. DOI: 10.1145/2405172.2405174.

[96] Hans Vatne Hansen et al. “Migration of Fine-grained Multimedia Applications.” In: *Proceedings of the Posters and Demo Track*. Ed. by Eric Wohlstadter. Middleware ’12. Montreal, Quebec, Canada: ACM, Dec. 2012, 12:1–12:2. ISBN: 978-1-4503-1612-5. DOI: 10.1145/2405153.2405165.

[97] Tobias Hoßfeld et al. “Quantification of YouTube QoE via Crowdsourcing.” In: *Proc. of ISM* (Dec. 2011), pp. 494–499. DOI: 10.1109/ISM.2011.87.

[98] Denis Howe, ed. *The Free On-line Dictionary of Computing*. http://foldoc.org/. [Online; accessed: 2018-09-16]. 2018.

[99] Wang Huan et al. “A Mechanism Based on Netfilter for Live TCP Migration in Cluster.” In: *Grid and Cooperative Computing (GCC), 9th International Conference on*. 2010, pp. 218–222.

[100] Markus C Huebscher and Julie A McCann. “A survey of autonomic computing—degrees, models, and applications.” In: *Computing Surveys* 40.3 (2008), pp. 7–28. DOI: 10.1145/1380584.1380585.

[101] Hulu. *System Requirements*. http://www.hulu.com/support/article/166380. [Online; accessed: 19-Aug-2017]. Feb. 2013.

[102] IBM Corporation. *Autonomic Computing Toolkit – Developer’s Guide*. https://www.ibm.com/developerworks/autonomic/books/fpy0mst.htm. [Online; accessed: 2018-09-23]. Aug. 2004.

[103] IBM Corporation. *Autononmic Computing Toolkit – Problem Determination Log/Trace Scenario Guide*. `https://www.ibm.com/developerworks/autonomic/books/fpv1scn.htm`. [Online; accessed: 2018-09-23]. Aug. 2004.

[104] Selim Ickin et al. "Factors influencing quality of experience of commonly used mobile applications." In: *Communications Magazine, IEEE* 50.4 (2012), pp. 48–56. DOI: `10.1109/MCOM.2012.6178833`.

[105] ITU. *ITU-T Recommendation G.114*. Tech. rep. International Telecommunication Union, 2003.

[106] Ramesh Jain. "Multimedia information retrieval: watershed events." In: *MIR '08: Proceedings of the 1st ACM international conference on Multimedia information retrieval*. ACM, Oct. 2008. DOI: `10.1145/1460096.1460135`.

[107] Ramesh Jain and Pinaki Sinha. "Content without context is meaningless." In: *Proceedings of the international conference on Multimedia*. ACM, 2010, pp. 1259–1268. ISBN: 978-1-60558-933-6. DOI: `10.1145/1873951.1874199`.

[108] D Jannach et al. "A knowledge-based framework for multimedia adaptation." In: *Applied Intelligence* 24.2 (Apr. 2006), pp. 109–125. DOI: `10.1007/s10489-006-6933-0`.

[109] Eric Jul et al. "Fine-grained mobility in the Emerald system." In: *ACM Transactions on Computer Systems* 6.1 (Feb. 1988), pp. 109–133. DOI: `10.1145/35037.42182`.

[110] Swaroop Kalasapur, Mohan Kumar, and Behrooz Shirazi. "Personalized Service Composition for Ubiquitous Multimedia Delivery." In: *Proceedings of the Sixth IEEE International Symposium on a World of Wireless Mobile and Multimedia Networks (WoWMoM'05)* (2005), pp. 1–6.

[111] Goran Karabeg. "Adaptation trigger mechanism." [Online `http://urn.nb.no/URN:NBN:no-45558`; accessed: 2018-09-09. MA thesis. University of Oslo, June 2014.

[112] Richard M. Karp. "Reducibility Among Combinatorial Problems." In: *50 Years of Integer Programming 1958-2008* (Nov. 2009), pp. 219–241. DOI: `10.1007/978-3-540-68279-0_8`.

[113] J.O. Kephart and D.M. Chess. "The vision of autonomic computing." In: *Computer* 36.1 (2003), pp. 41–50. ISSN: 0018-9162. DOI: 10.1109/mc.2003.1160055.

[114] R. Kernchen et al. "Intelligent Multimedia Presentation in Ubiquitous Multidevice Scenarios." In: *MultiMedia, IEEE* (2010).

[115] Michael Kerrisk. *shm_overview(7) - overview of the POSIX shared memory*. http://man7.org/linux/man-pages/man7/shm_overview.7.html. [Online; accessed: 2018-06-30]. Dec. 2016.

[116] George Kiagiadakis. *ipcpipeline: Splitting a GStreamer pipeline into multiple processes*. https://www.collabora.com/news-and-blog/blog/2017/11/17/ipcpipeline-splitting-a-gstreamer-pipeline-into-multiple-processes/. [Online; accessed on: 2018-09-30]. Nov. 2017.

[117] Bryan Knutz and Karthik Rajan. "MIGSOCK: Migratable TCP Socket in Linux." MA thesis. Carnegie Mellon University. Information Networking Institute, 2002.

[118] Bernhard Korte and Jens Vygen. *Combinatorial Optimization*. Theory and Algorithms. Springer Science & Business Media, Jan. 2012. ISBN: 3642244882.

[119] Rainer Koster et al. "Infopipes for Composing Distributed Information Flows." In: *Proc. of M3W*. Ottawa, Ontario, Canada, 2001, pp. 44–47. ISBN: 1-58113-396-0. DOI: 10.1145/985135.985150.

[120] M Kumar et al. "PICO: a middleware framework for pervasive computing." In: *Pervasive Computing, IEEE* 2.3 (2003), pp. 72–79. DOI: 10.1109/MPRV.2003.1228529.

[121] J Lachner et al. "Challenges Toward User-Centric Multimedia." In: *Semantic Media Adaptation and Personalization, Second International Workshop on* (2007), pp. 159–164. DOI: 10.1109/SMAP.2007.35.

[122] Oussama Layaida and Daniel Hagimont. "Designing Self-adaptive Multimedia Applications Through Hierarchical Reconfiguration." In: *Proc. of Distributed Applications and Interoperable Systems (DAIS)*. 2005, pp. 95–107. ISBN: 978-3-540-31582-7. DOI: 10.1007/11498094_9.

[123] Yong-Ju Lee et al. *UMOST : Ubiquitous Multimedia Framework for Context-Aware Session Mobility*. IEEE, 2008. ISBN: 978-0-7695-3134-2. DOI: 10.1109/MUE.2008.120.

[124] Tobin J. Lehman et al. "Hitting the distributed computing sweet spot with TSpaces." In: *Comput. Netw.* 35.4 (Mar. 2001), pp. 457–472.

[125] Klaus Leopold, Dietmar Jannach, and Hermann Hellwagner. "A Knowledge and Component Based Multimedia Adaptation Framework." In: *Proceedings of the IEEE Sixth International Symposium on Multimedia Software Engineering*. IEEE Computer Society, 2004, pp. 10–17. ISBN: 0-7695-2217-3.

[126] Anany Levitin. *Introduction to the Design and Analysis of Algorithms*. ISBN 9780132316811. Pearson Education, Dec. 2011, pp. 172–174. ISBN: 9780132316811.

[127] Chi-Yu Li et al. "A multimedia service migration protocol for single user multiple devices." In: *Communications (ICC), 2012 IEEE International Conference on*. June 2012, pp. 1923–1927. DOI: 10.1109/ICC.2012.6363673.

[128] K. Li. "Shared virtual memory on loosely coupled multiprocessors." PhD thesis. Yale University, Jan. 1986.

[129] Kai Li and Paul Hudak. "Memory Coherence in Shared Virtual Memory Systems." In: *ACM Trans. Comput. Syst.* 7.4 (Nov. 1989), pp. 321–359. ISSN: 0734-2071. DOI: 10.1145/75104.75105.

[130] Ning Li et al. *Device and service descriptions for ontology-based ubiquitous multimedia services*. ACM, Nov. 2008. ISBN: 978-1-60558-269-6. DOI: 10.1145/1497185.1497265.

[131] Sten A Lundesgaard, Ketil Lund, and Frank Eliassen. "Utilising Alternative Application Configurations in Context- and QoS-Aware Mobile Middleware." In: *Distributed Applications and Interoperable Systems*. Springer Berlin Heidelberg, June 2006, pp. 228–241. ISBN: 978-3-540-35126-9. DOI: 10.1007/11773887_18.

[132] Leszek. Maciaszek. *Requirements analysis and system design / Leszek A. Maciaszek*. English. 3rd ed. Addison-Wesley Harlow, 2007, xxxvii, 612 p. : ISBN: 9780321440365.

[133] Andrew Makhorin. *CNF Satisfiability Problem*. Linux package glpk-doc. Aug. 2011.

[134] Andrew Makhorin. *GNU Linear Programming Kit – Graph and Network Routines*. Linux package glpk-doc. Mar. 2016.

[135] Andrew Makhorin. *GNU Linear Programming Kit – Reference Manual for GLPK version 4.64*. Linux package glpk-doc. Draft. Nov. 2017.

[136] Andrew Makhorin. *Modeling Language GNU MathProg – Language Reference for GLPK version 4.58*. Linux package glpk-doc. Draft. Feb. 2016.

[137] D A Maltz and P Bhagwat. "MSOCKS: an architecture for transport layer mobility." In: *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE.* Mar. 1998, 1037–1045 vol.3. DOI: `10.1109/INFCOM.1998.662913`.

[138] J Manner and M Kojo. *RFC 3753 - Mobility Related Terminology*. Tech. rep. Network Working Group, June 2004.

[139] *MIDDLEWARE '12: Proceedings of the 9th Middleware Doctoral Symposium of the 13th ACM/IFIP/USENIX International Middleware Conference*. Montreal, Quebec, Canada: ACM, 2012. ISBN: 978-1-4503-1611-8.

[140] Dejan S Milojičić et al. "Process migration." In: *ACM Computing Surveys* 32.3 (2000), pp. 241–299. DOI: `10.1145/367701.367728`.

[141] Netflix. *Internet Connection Speed Recommendations*. `https://help.netflix.com/en/node/306`. [Online; accessed: 2018-08-18]. 2013.

[142] Binh Nguyen. *Linux Filesystem Hierarchy*. `http://tldp.org/guides.html`. Version 0.65. [Online; accessed: 2018-06-30]. July 2004.

[143] Eila Niemel a and Juhani Latvakoski. "Survey of requirements and solutions for ubiquitous software." In: *Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia*. ACM, 2004, pp. 71–78. ISBN: 1-58113-981-0. DOI: `10.1145/1052380.1052391`.

[144] Hyeong-Seok Oh et al. "Evaluation of Android Dalvik virtual machine." In: *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems - JTRES '12* (2012). DOI: `10.1145/2388936.2388956`.

[145] Brian Oki et al. "The Information Bus: An Architecture for Extensible Distributed Systems." In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. SOSP '93. Asheville, North Carolina, USA: ACM, 1993, pp. 58–68. ISBN: 0-89791-632-8. DOI: 10.1145/168619.168624.

[146] Manuel Ortega-Moral, Ignacio Peinado, and Gregg C Vanderheiden. "Cloud4all: Scope, Evolution and Challenges." In: *Universal Access in Human-Computer Interaction. Design for All and Accessibility Practice*. Springer International Publishing, June 2014, pp. 421–430. ISBN: 978-3-319-07508-2. DOI: 10.1007/978-3-319-07509-9_40.

[147] Steven Osman et al. "The design and implementation of Zap: a system for migrating computing environments." In: *SIGOPS Oper. Syst. Rev.* 36.SI (2002), pp. 361–376. DOI: 10.1145/844128.844162.

[148] Shumao Ou, Kun Yang, and Jie Zhang. "An effective offloading middleware for pervasive services on mobile devices." In: *Pervasive Mob. Comput.* 3.4 (2007), pp. 362–385. DOI: 10.1016/j.pmcj.2007.04.004.

[149] P Pantazopoulos, M Karaliopoulos, and I Teletraffic Congress ITC 2011 23rd International Stavrakakis. "Centrality-driven scalable service migration." In: *Teletraffic Congress (ITC), 2011 23rd International* (2011).

[150] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., 1982. ISBN: 0-13-152462-3.

[151] Vangelis Th Paschos. *Paradigms of Combinatorial Optimization*. Problems and New Approaches. John Wiley & Sons, Feb. 2013. ISBN: 1118600274.

[152] Vangelis Th. Paschos. *Concepts of Combinatorial Optimization*. John Wiley & Sons, Dec. 2012. ISBN: 1118600231.

[153] Nearchos Paspallis and George A Papadopoulos. "A pluggable middleware architecture for developing context-aware mobile applications." In: *Personal and Ubiquitous Computing* 18.5 (June 2014), pp. 1099–1116. DOI: 10.1007/s00779-013-0722-7.

[154] Havoc Pennington et al. *D-Bus Specification*. https://dbus.freedesktop.org/doc/dbus-specification.html. [Online; accessed: 2018-06-30]. Mar. 2018.

[155] F Pereira and I Burnett. “Universal multimedia experiences for tomorrow.” In: *Signal Processing Magazine, IEEE* 20.2 (2003), pp. 63–73. DOI: 10.1109/MSP.2003.1184340.

[156] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. “LIME: Linda Meets Mobility.” In: *Proceedings of the 21st International Conference on Software Engineering*. ICSE ’99. Los Angeles, California, USA: ACM, May 1999, pp. 368–377. ISBN: 1-58113-074-0. DOI: 10.1145/302405.302659.

[157] Raffaele Quitadamo. “The Issue of Strong Mobility: an Innovative Approach based on the IBM Jikes Research Virtual Machine.” PhD thesis. University of Modena and Reggio Emilia, Apr. 2008.

[158] Raffaele Quitadamo, Giacomo Cabri, and Letizia Leonardi. “Mobile JikesRVM: A framework to support transparent Java thread migration.” In: *Sci. Comput. Program.* 70.2-3 (2008), pp. 221–240. DOI: 10.1016/j.scico.2007.07.009.

[159] Raffaele Quitadamo et al. “The PIM: an innovative robot coordination model based on Java thread migration.” In: *Proceedings of the 6th international symposium on Principles and practice of programming in Java*. ACM, 2008, pp. 43–51. ISBN: 978-1-60558-223-8. DOI: 10.1145/1411732.1411739.

[160] Tim Rahrer, Riccardo Fiandra, and Steven Wright. *Triple-play Services Quality of Experience (QoE) Requirements*. Tech. rep. TR-126. DSL Forum, Dec. 2006.

[161] Haakon Wilhelm Ravik. “A Real-Time Video Retargeting Plugin for GStreamer.” [Online http://urn.nb.no/URN:NBN:no-56335; accessed: 2018-09-09]. MA thesis. Oslo, Norway: University of Oslo, Sept. 2016.

[162] D Romero. “Context-aware middleware: An overview.” In: *Paradigma* (2008).

[163] Romain Rouvoy et al. “MUSIC: an autonomous platform supporting self-adaptive mobile applications.” In: *MobMid ’08: Proceedings of the 1st workshop on Mobile middleware: embracing the personal communication device*. ACM, Dec. 2008.

[164] Giovanni Russello, Michel Chaudron, and Maarten van Steen. *GSpace : Tailorable Data Distribution in Shared Data Space Systems*. Tech. rep. Technische Universiteit Eindhoven, 2004.

[165] J H Saltzer, D P Reed, and D D Clark. "End-to-end arguments in system design." In: *ACM Transactions on Computer Systems* 2.4 (1984), pp. 277–288. DOI: 10.1145/357401.357402.

[166] M. Satyanarayanan. "Pervasive computing: vision and challenges." In: *IEEE Personal Communications* 8.4 (2001), pp. 10–17. ISSN: 1070-9916. DOI: 10.1109/98.943998.

[167] Mihaela van der Schaar and Philip A Chou. *Multimedia over IP and Wireless Networks*. Compression, Networking, and Systems. Academic Press, July 2011. ISBN: 9780080474960.

[168] Ulrich Scholz and Stephan Mehlhase. "Co-ordinated Utility-Based Adaptation of Multiple Applications on Resource-Constrained Mobile Devices." In: *Proc. of DAIS*. Ed. by Frank Eliassen and Rüdiger Kapitza. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 198–211. ISBN: 978-3-642-13645-0. DOI: 10.1007/978-3-642-13645-0_15.

[169] Merat Shahidi, Ning Li, and A Hamid Aghvami. "Selection algorithm for multimedia adaptation mechanisms in ubiquitous service environments." In: *iiWAS '10: Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*. 2010.

[170] Merat Shahidi, Nika Naghavi, and A. Hamid Aghvami. "Content adaptor selection models in Adaptation Management Framework." In: *2011 18th International Conference on Telecommunications* (May 2011). DOI: 10.1109/cts.2011.5898971.

[171] Jalal Al-Muhtadi Shiva Chetan. "Mobile Gaia: A Middleware for Ad-hoc Pervasive Computing." In: *Consumer Communications and Networking Conference*. IEEE, 2005, pp. 223–228. ISBN: 0-7803-8784-8. DOI: 10.1109/CCNC.2005.1405173.

[172] J. Siegel. *CORBA 3 fundamentals and programming*. Wiley computer publishing. John Wiley & Sons, 2000. ISBN: 9780471295181.

[173] José Simão, Tiago Garrochinho, and Luís Veiga. “A checkpointing-enabled and resource-aware Java Virtual Machine for efficient and robust e-Science applications in grid environments.” In: *Concurrency and computation: Practice and experience* 24.13 (2012), pp. 1421–1442. DOI: 10.1002/cpe.1879.

[174] Skype. *How much bandwidth does Skype need?* https://support.skype.com/en/faq/FA1417/how-much-bandwidth-does-skype-need. [Online; accessed on: 2018-08-18].

[175] Peter Smith and Norman C. Hutchinson. “Heterogeneous process migration: the Tui system.” In: *Software: Practice and Experience* 28.6 (May 1998), pp. 611–639. ISSN: 1097-024X.

[176] Jo a o Pedro Sousa and David Garlan. “Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments.” In: *Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*. Deventer, The Netherlands, The Netherlands: Kluwer, B.V., 2002, pp. 29–43. ISBN: 1-4020-7176-0.

[177] R. Steinmetz. “Human perception of jitter and media synchronization.” In: *IEEE Journal on Selected Areas in Communications* 14.1 (Jan. 1996), pp. 61–72. DOI: 10.1109/49.481694.

[178] Peter Norvig Stuart Russell. *Artificial Intelligence. A Modern Approach [Global Edition]*. 3rd. Pearson, 2010. ISBN: 9781292153964.

[179] F Sultan et al. “Migratory TCP: Connection Migration for Service Continuity in the Internet.” In: *22nd International Conference on Distributed Computing Systems*. IEEE Comput. Soc, 2002, pp. 469–470. ISBN: 0-7695-1585-1. DOI: 10.1109/ICDCS.2002.1022294.

[180] A S Tanenbaum. *Distributed Operating Systems*. Prentice-Hall international editions. Prentice Hall, 1995. ISBN: 9780131439344.

[181] Andrew S. Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson Prentice-Hall, 2015. ISBN: [9780133591620].

[182] The Austing Group. “IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R)).” In: *IEEE Std 1003.1, 2004 Edition The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-*

2002 and IEEE Std 1003.1-2001/Cor 2-2004. Shell (Dec. 2008), pp. 1–3874. DOI: 10.1109/IEEESTD.2008.7394902.

[183] The GNOME Project. *GIO Reference Manual for GIO 2.56.1*. https://developer.gnome.org/gio/2.56/ch01.html. [Online; accessed: 2018-06-30]. 2016.

[184] The GNOME Project. *GLib Reference Manual for GLib 2.56.1*. https://developer.gnome.org/glib/2.56/glib.html. [Online; accessed: 2018-06-30]. 2018.

[185] *The Jargon File 4.4.7*. http://www.catb.org/jargon/html/. Version 4.4.7. [Online; accessed: 2018-04-26]. Dec. 2003.

[186] *Three Minute Thesis (3MT) competition of the IEEE International Conference on Multimedia and Expo (ICME)*. [Online http://www.icme2018.org/student_participation; accessed: 2018-08-12]. San Diego, California, USA: IEEE, 2018.

[187] TRAMP Project. *TRAMP Real-time Application Mobility Platform*. World Wide Web. http://tramp-project.org/. Aug. 2012.

[188] Gareth Tyson et al. “Juno: A Middleware Platform for Supporting Delivery-Centric Application.” In: *ACM Transactions on Internet Technology* V (Jan. 2012).

[189] Justin Uberti and Peter Thatcher. *WEB Real Time Communication (WWW, W3c), "WebRTC"*. https://webrtc.org/. [Online; accessed: 2018-08-12]. May 2017.

[190] Francisco Javier Velazquez-Garcia. “DAMPAT: Dynamic Adaptation of Multimedia Presentations in Application Mobility.” In: *GStreamer Conference*. [Online https://gstreamer.freedesktop.org/conference/2017; accessed: 2018-08-12]. Prague, Czech Republic, Oct. 2017.

[191] Francisco Javier Velázquez-García and Frank Eliassen. “DAMPAT: Dynamic Adaptation of Multimedia Presentations in Application Mobility.” In: *Proc. of International Symposion on Multimedia (ISM)*. Dec. 2017, pp. 312–317. DOI: 10.1109/ISM.2017.56.

[192] Francisco Javier Velázquez-García et al. “Autonomic Adaptation of Multimedia Content Adhering to Application Mobility.” In: *Distributed Applications and Interoperable Systems*. Ed. by Silvia Bonomi and Etienne Rivière. Madrid, Spain: Springer International Publishing, 2018, pp. 153–168. ISBN: 978-3-319-93767-0. DOI: `10.1007/978-3-319-93767-0_11`.

[193] Francisco Javier Velázquez-García et al. “Dynamic Adaptation of Multimedia Presentations for Videoconferencing in Application Mobility.” In: *International Conference on Multimedia and Expo (ICME)*. San Diego, California, USA, July 2018. DOI: `10.1109/ICME.2018.8486565`.

[194] Francisco Javier Velázquez-García et al. “SOCKMAN: Socket Migration for Multimedia Applications.” In: *The 12th International Conference on Telecommunications (ConTEL)*. Ed. by K. Pripužić and M. Banek. Zagreb, Croatia, June 2013, pp. 115–122. ISBN: 978-953-184-180-1.

[195] S. Vinoski. “CORBA: integrating diverse applications within distributed heterogeneous environments.” In: *IEEE Communications Magazine* 35.2 (1997), pp. 46–55. ISSN: 0163-6804. DOI: `10.1109/35.565655`.

[196] Marc Weiser. “The computer for the 21st Century.” In: *IEEE Pervasive Computing* 99.1 (Sept. 1991), pp. 19–25. DOI: `10.1109/MPRV.2002.993141`.

[197] Marc Weiser, R Gold, and J S Brown. “The origins of ubiquitous computing research at PARC in the late 1980s.” In: *IBM Systems Journal* 38.4 (1999), pp. 693–696. DOI: `10.1147/sj.384.0693`.

[198] Mark Weiser and John Seely Brown. “The Coming Age of Calm Technology.” In: *Beyond Calculation* (1997), pp. 75–85. DOI: `10.1007/978-1-4612-0685-9_6`.

[199] Haakon Wilhelm Ravik and Francisco Javier Velazquez-Garcia. “Gst-SeamCrop Real-time video retargeting in Nvidia GPU.” In: *GStreamer Conference*. [Online `https://gstreamer.freedesktop.org/conference/2016`; accessed: 2018-08-12]. Berlin, Germany, Oct. 2016.

[200] Christian Wimmer et al. “Maxine: An approachable virtual machine for, and in, java.” In: *Transactions on Architecture and Code Optimization (TACO)* 9.4 (Jan. 2013). DOI: `10.1145/2400682.2400689`.

[201] Min Xu, Jesse S Jin, and Suhuai Luo. *Personalized video adaptation based on video content analysis*. ACM, Aug. 2008. ISBN: 978-1-60558-261-0. DOI: 10.1145/1509212.1509216.

[202] S.S. Yau et al. “Reconfigurable context-sensitive middleware for pervasive computing.” In: *IEEE Pervasive Computing* 1.3 (July 2002), pp. 33–40. ISSN: 1536-1268. DOI: 10.1109/mprv.2002.1037720.

[203] Taewan You and Seungyun Lee. “The Framework for Mobility and Multihoming Using Overlay Network.” In: *8th International Conference on Advanced Communication Technology*. IEEE, pp. 1803–1806. ISBN: 89-5519-129-4. DOI: 10.1109/ICACT.2006.206340.

[204] Ping Yu et al. “Application mobility in pervasive computing: A survey.” In: *Pervasive and Mobile Computing* 9.1 (2013), pp. 2–17. ISSN: 1574-1192. DOI: 10.1016/j.pmcj.2012.07.009.

[205] Zhiwen Yu et al. “An OSGi-based infrastructure for context-aware multimedia services.” In: *Communications Magazine, IEEE* 44.10 (2006), pp. 136–142. DOI: 10.1109/MCOM.2006.1710425.

[206] Zhiyong Yu et al. “Toward an Understanding of User-Defined Conditional Preferences.” In: *Dependable, Autonomic and Secure Computing, 2009. DASC ’09. Eighth IEEE International Conference on*. 2009, pp. 203–208. DOI: 10.1109/DASC.2009.52.

[207] Victor C Zandy and Barton P Miller. “Reliable Network Connections.” In: *the 8th annual international conference*. ACM Press, 2002, p. 95. ISBN: 158113486X. DOI: 10.1145/570645.570657.

[208] Ying Zhang et al. “Refactoring android Java code for on-demand computation offloading.” In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. ACM, 2012, pp. 233–248. ISBN: 978-1-4503-1561-6. DOI: 10.1145/2384616.2384634.