Autonomic Adaptation of Multimedia Content Adhering to Application Mobility

Francisco Javier Velázquez-García, Pal Halvorsen, Håkon Kvale Stensland, Frank Eliassen

Autonomic adaptation of multimedia content adhering to application mobility

Francisco Javier Velázquez-García[1,2], Pål Halvorsen[1,2], Håkon Kvale Stensland[1,2], and Frank Eliassen[2]

[1] Simula Research Laboratory, P.O. Box 134, 1325, Lysaker, Norway
[2] Department of Informatics, University of Oslo, Oslo, Norway
{francisv,paalh,haakonks,frank}@ifi.uio.no

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.

Keywords: self-configuration; self-adaptive; self-optimization; self-awareness; application mobility; multimedia pipelines, MAPE-K, DSPL

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 [14]. 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 [9], 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 [6]. 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 con-

figuration that produces the highest utility according to the current contextual situation and user preferences.

DAMPAT follows the Dynamic Software Product Lines (DSPL) engineering approach [2]. 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 [13] and [12] 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 [10]), in the application mobility paradigm.

In the remainder of the paper, Section 2 explains the main challenges of design and implementation decisions of the proposed system. Section 3 evaluates the parts of the system that can negatively impact the seamlessness of multimedia mobile applications. Section 4 compares DAMPAT with related work. Finally, Section 5 concludes the paper.

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 [7], where K is the knowledge created and used across the MAPE phases (see Fig. 1). Next, we describe in a top-down manner how the MAPE-K loop is applied in DAMPAT.

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

Fig. 1 represents an *autonomic manager*, a *managed element*, *sensors*, and *effectors*. The autonomic manager is a software component configured by human

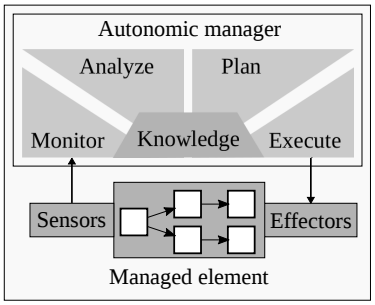

Fig. 1. Structure of Monitor, Analyze, Plan, and Execute (MAPE)-K control loop

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 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 2.2 to 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 Fig. 1 carry out changes to the managed element.

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.

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 [1] to aggregate the monitored data of all the surrounding devices to which an application can move.

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.

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.

Fig. 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

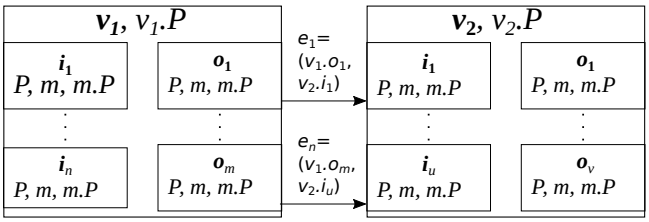

Fig. 2. Multigraph that shows vertices $v_1, v_2 \in V$ representing pipeline components. P represent a set of properties, i and o represent input and output connectors, $m \in M$ the supported modalities by the connectors, and $e \in E$ represent links or pipes between connectors.

the multimedia pipeline model represented in Fig. 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 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 Fig. 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 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 [4] is the only free and open source, multi-platform, multimedia framework actively implementing and maintaining these mechanisms. Therefore, we leverage GStreamer pipelines in DAMPAT.

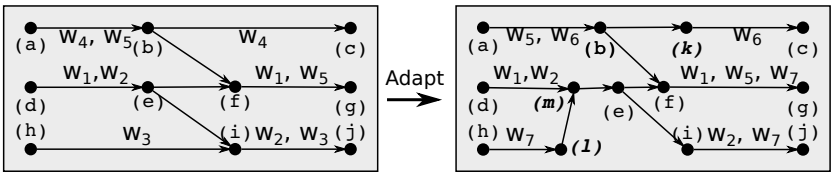

Fig. 3. Graph abstraction of multimedia pipeline of one videoconferencing peer before and after adaptation. On the left, pipeline that consumes and produces video and audio. On the right, pipeline that consumes video and text from a peer that cannot process audio, this pipeline allows its user to produce 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.

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 Fig. 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.

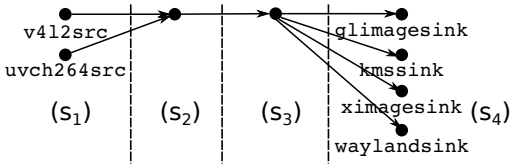

Fig. 4. Example of functional path (w) that captures video from a webcam and renders it in a display. In this example, w has four functional stages ($\{s\}_1^4$) and eight possible path combinations.

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 [13].

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 1.1 and 1.2 show snippets of the configuration file for one functional path in Fig. 4.

Listing 1.1. Snippet 1 of `w1.conf`

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

Listing 1.2. Snippet 2 of `w1.conf`

```
6 [vertex videosrc]
7 name=v4lsrc0
8 output-conn=v4lsrc0.src.0
9 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 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 1.1 and 1.2, 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 [6]. 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 [13] 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. Fig. 5 illustrates this analogy in a oversimplified pipeline that processes video and audio modalities.

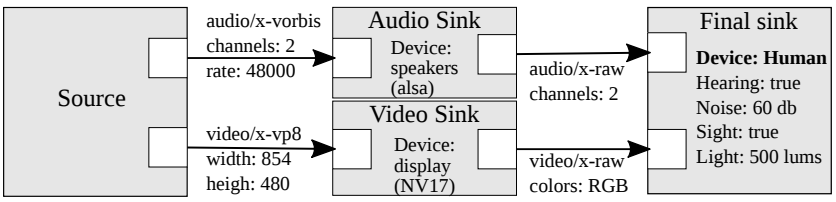

Fig. 5. Oversimplified pipeline to make an analogy of a human consumer as a component in a multimedia pipeline. In this analogy, the input connectors (interfaces) of a human consumer are the hearing and sight senses. DAMPAT selects the pipeline variant with connectors that are compatible with the available and appropriate user interfaces in a given context.

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) 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 $\Upsilon(u, g) = \sum_{j=1}^{l} ut(u.p_j, g.p_j) \cdot u.p_j.we$ [13].

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.

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 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.

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 [13], and [12].

For completeness, we briefly describe both prototypes and the experiments. In evaluation 1 [13], 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 [12], 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 Fig. 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.

3.1 Plan phase

In this section, we discuss results from our previous work ([13] and [12]) 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 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 [11], should be implemented in DAMPAT to tackle larger search spaces. The implementation of greedy techniques, however, is left as future work.

3.2 Execution phase

In this section, we discuss results from our previous work ([13] and [12]) 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 Fig. 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 Fig. 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 cre-

ation of the variability search space using the reconfiguration mechanisms are future work.

4 Related Work

MUSIC [5] 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. [5] recognized that support for multimedia content adaptation in a challenging research alley, and left it as future work.

PLASMA [8] 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 2.4), and a reconfiguration model, similar to the Execute phase (Section 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 [3] 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.

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.

Bibliography

[1] Andic, M.: Negotiation and Data Transfer for Application Mobility. Master's thesis, University of Oslo (2015)

[2] Bashari, M., Bagheri, E., Du, W.: Dynamic Software Product Line Engineering: A Reference Framework. International Journal of Software Engineering and Knowledge Engineering **27**(2), 191–234 (2017). https://doi.org/10.1142/S0218194017500085

[3] Black, A.P., Huang, J., Koster, R., Walpole, J., Pu, C.: Infopipes: An abstraction for multimedia streaming. Multimedia Systems **8**(5), 406–419 (2002). https://doi.org/10.1007/s005300200062

[4] GStreamer community: GStreamer Open Source Multimedia Framework. https://gstreamer.freedesktop.org/, [Online; accessed: 2018-03-29]

[5] Hallsteinsen, S., Geihs, K., Paspallis, N., Eliassen, F., Horn, G., Lorenzo, J., Mamelli, A., Papadopoulos, G.: A development framework and methodology for self-adapting applications in ubiquitous computing environments. Journal of Systems and Software **85**(12), 2840–2859 (2012). https://doi.org/10.1016/j.jss.2012.07.052

[6] Huebscher, M.C., McCann, J.A.: A survey of autonomic computing—degrees, models, and applications. Computing Surveys **40**(3), 7–28 (2008). https://doi.org/10.1145/1380584.1380585

[7] Kephart, J., Chess, D.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003). https://doi.org/10.1109/mc.2003.1160055

[8] Layaida, O., Hagimont, D.: Designing self-adaptive multimedia applications through hierarchical reconfiguration. In: Proc. of DAIS. pp. 95–107 (2005)

[9] Milojičić, D.S., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process migration. ACM Computing Surveys **32**(3), 241–299 (2000). https://doi.org/10.1145/367701.367728

[10] Ravik, H.W.: A Real-Time Video Retargeting Plugin for GStreamer. Master's thesis, University of Oslo (Sep 2016)

[11] Scholz, U., Mehlhase, S.: Co-ordinated utility-based adaptation of multiple applications on resource-constrained mobile devices. In: Proc. of DAIS. pp. 198–211 (2010)

[12] Velázquez-García, F.J., Halvorsen, P., Stensland, H.K., Eliassen, F.: Dynamic adaptation of multimedia presentations for videoconferencing in application mobility (2018), to appear in: Proc. of ICME

[13] Velázquez-García, F.J., Eliassen, F.: DAMPAT: Dynamic Adaptation of Multimedia Presentations in Application Mobility. In: Proc. of ISM. pp. 312–317 (2017). https://doi.org/10.1109/ISM.2017.56

[14] Yu, P., Ma, X., Cao, J., Lu, J.: Application mobility in pervasive computing: A survey. Pervasive and Mobile Computing **9**(1), 2 – 17 (2013). https://doi.org/10.1016/j.pmcj.2012.07.009