Designing Self-adaptive Multimedia Applications Through Hierarchical Reconfiguration

Oussama Layaida and Daniel Hagimont

SARDES Project, INRIA Rhône-Alpes
First.last@inria.fr

Abstract. Distributed multimedia applications are very sensitive to resource variations. An attractive way for dealing with dynamic resource variations consists in making applications adaptive, and even self-adaptive. The objective is to grant applications the ability to observe themselves and their environment, to detect significant changes and to adjust their behavior accordingly. This issue has been the subject of several works; however the proposed solutions lack flexibility and a high-level support that eases the development of adaptive applications. This paper presents PLASMA, a component-based framework for building multimedia applications. PLASMA relies on a hierarchical composition and reconfiguration model which provides the expected support. The experimental evaluation shows that adaptation can be achieved with a very low overhead, while significantly improving QoS of multimedia applications as well as resource usage on mobile equipments.

1 Introduction

Recent advances in mobile equipments and wireless networking have led to the emergence of a wide range of peripherals such as, Personal Digital Assistant (PDA), hand-held computers, Smart Phones, eBooks, etc. The Internet infrastructure became, like never before, heterogeneous and dynamic. System and network resources such as network bandwidth, CPU load or battery life time are characterized by unpredictable variations making difficult to guarantee the correct execution of multimedia applications.

The most attractive approach to deal with this issue consists in making these applications self-adaptive, that is, grant them the ability to observe themselves and their environment, to detect significant changes and adapt their own behavior in QoS-specific ways. A well recognized approach to achieve it is the use of component-based technologies [1]. The common idea consists in implementing multimedia-related functions in separate components. Various multimedia services can then be built by selecting and assembling the appropriate ones within the same application. Likewise, adaptation is achieved by means of high-level component reconfigurations such as: adjusting component properties, stopping/starting a subset of components, removing/inserting components or modifying their assembly. Complex operations can be made-up of combination of those basic operations, performed in an appropriate order.

L. Kutvonen and N. Alonistioti (Eds.): DAIS 2005, LNCS 3543, pp. 95 – 107, 2005.

This article describes PLASMA, a component-based framework for the development of self-adaptive multimedia applications. PLASMA relies on an advanced component model [2], whose main features are: recursive composition and hierarchical reconfiguration management. This allows to model reconfiguration in a generic way thus addressing arbitrary applications and adaptation policies. The remainder of the article is organized as follows: Next section presents a classification of related works. Section 3 introduces our design choices and the PLASMA architecture. Section 4 presents an application use case and a performance evaluation of PLASMA. Finally, section 5 concludes this paper and presents perspectives.

2 Related Work

As previously mentioned, adaptivity is tightly linked with component-based technologies. Work around multimedia applications has led to several component-based frameworks such as DirectShow (Microsoft) [6] JMF (Sun) [13] and PREMO (ISO) [7], easing the development of multimedia applications. Following this vein, the advantages of component-based technologies have motivated several research works in order to bring adaptivity to multimedia applications [3, 15, 14] but very few of them considering run-time reconfiguration. This feature has been addressed with different approaches:

- ***Static Reconfiguration Policies***: A first approach to reconfiguration uses static reconfiguration policies to deal with specific resource variations. VIC [17] is a well-known example of such applications. Although it is not component-based, it uses the RTCP [10] feedback mechanism and a loss-based congestion control algorithm [11] that adapts media streams to the available bandwidth. Reconfiguration operations consist in tuning key encoding properties (quality factor, frame rate, etc.) in order to adjust the transmission rate appropriately. Nevertheless, the use of static reconfigurations is too restrictive as they have to be anticipated at development-time.
- ***Component-Based Frameworks with Reconfiguration Capabilities***: Some component-based frameworks grant application developers with enhanced reconfiguration capabilities. The Toolkit for Open Adaptive Streaming Technology (TOAST) [8] investigates the use of open implementation and reflection to ease the development of adaptation strategies. However, it remains to the responsibility of the application developer to deal with resource and application monitoring, reconfiguration decisions and their implementation, which is a pretty heavy task.
- ***Component-Embedded Reconfiguration Policies***: To fill this gap, some works propose to integrate reconfiguration features in the functional components themselves. In Microsoft DirectShow [6] for example, processing components (called filters) exchange *in-stream QoS messages* traveling in the opposite direction of the data flow. Using this mechanism a component may indicate to its predecessors that data is being produced too rapidly (*a flood*) or too slowly (*a famine*), which decrease (or increase) their data processing rate in response. Such mechanism can be easily extended to support a larger scope of QoS control, as

proposed in [5, 19]. However, the limitation of this approach is that reconfigurations only occur in the scope of each component rather than in that of the application. Although it is possible to change the behavior of a given component, it is not possible to perform a component replacement.

- ***Separate Reconfiguration Manager***: By opposition to the previous approach, some works have proposed that all reconfiguration features be integrated in separate managers. Instead of sending QoS messages through components, they are delivered to a reconfiguration manager, which performs reconfiguration operations. This approach is applied in the *DaCapo++* [12] communication framework, in the *2KQ* framework [16] for resource management and in CANS (Composable Adaptive Network Services) [9] and APC (Automatic Path Creation Service) [18] to design adaptive media transcoding gateways. The main limitation of this approach is that such a manager is tightly-coupled with the targeted application, and especially its architecture. Any modification of the application architecture requires an update of the manager's implementation[1].

3 Component Architecture

The previous section has shown the limitation of previous approaches in addressing requirements mentioned before. This limitation resides in the use of flat component models. Indeed, the integration of reconfiguration at the component level limits reconfiguration capabilities. At the other hand, making it separated from the application requires strong assumptions on its structure. Hence, a new component model becomes necessary with the goal of addressing a large scope, or better, all possible adaptation algorithms within arbitrary component assemblies. This section describes how this requirement is addressed in PLASMA.

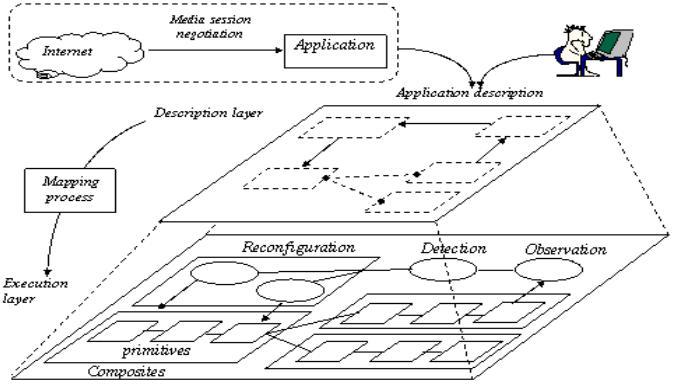

Fig. 1. Overall architecture design

[1] Or it should be explicitely managed in the manager’s implementation.

3.1 Design Choices

As depicted in Figure 1, the architecture of PLASMA is composed of: a *description layer* providing tools for the description of applications and their adaptation policies, and an *execution layer* for the composition and the execution of applications.

- *Using a Dynamic ADL*: An application is described using a dynamic ADL (Architecture Description Language). It offers constructs for the description of the application architecture as well as its adaptation policy in terms of observations, detection of relevant changes and processing of reconfiguration operations. All these operations are modeled within the language in a generic way so that it is possible to describe every possible application and its adaptation policies. Descriptions can be written by developers as configuration files or automatically generated by applications requesting specific multimedia services. The framework provides tools to translate a given description into the appropriate component assembly in the execution level. A detailed example of the ADL language is presented in section 4.
- *Recursive Composition*: The construction of applications is facilitated by a recursive component model. An application is divided into several parts, called composite components, which are in turn defined with sub-components. The model allows arbitrary number of levels; the lowest includes primitive components encapsulating functional code. This model is based on the Fractal composition framework [2].
- *Hierarchical Reconfiguration Management*: Reconfiguration policies can be defined at any level of the component hierarchy. Each component has its own reconfiguration manager responsible for reconfiguring its content. In a primitive component, this manager acts on its functional code, for instance by modifying parameters. In a composite component, it applies reconfiguration on its sub-components, transparently to higher levels. This model allows dividing reconfiguration responsibilities into several hierarchical managers, each dealing with a specific part of the application.
- *Composable Adaptation Policies*: Adaptation operations require mainly: the observation of application and resource states, the detection of relevant changes and activation of reconfiguration operations. These functions are implemented in PLASMA as separate and reusable components that can be used to compose every possible adaptation policy.

3.2 Component Architecture

The PLASMA component architecture encompasses three kinds of components: *Media*, *Monitoring* and *Reconfiguration* components. The following subsections detail the role of each of them.

3.2.1 Media Components

Media components represent the computational units used for the composition of the multimedia applications. An application is decoupled into three hierarchical levels, each providing a specific functionality:

- *Media Primitive* (MP) components are the lowest-level processing entities. They implement basic multimedia-related functions such as MPEG decoding, H.261 encoding, UDP transmission, etc.
- *Media Composite* (MC) components are composite components which represent higher-level functions such as: Decoding, Network Transmission, etc. Each media composite deals with a group of MPs and is responsible for their creation, configuration and reconfiguration, transparently to its outside. In Figure 2, an *InputStream* composite is composed of three Media Primitives in the case of an RTP input stream: an RTP receiver, a *Demultiplexer* (Demux) to separate multiple streams and a Depacketizer to reconstitute media data. On the other hand, an HTTP input stream requires one primitive component: *HTTP-Receiver*.
- The *Media-Session* (MS) component is a composite which encapsulates MCs. The Media-Session represents an application configuration and exposes all control features that can be made upon it (i.e. VCR-like operations: start, pause, stop, forward, etc.).

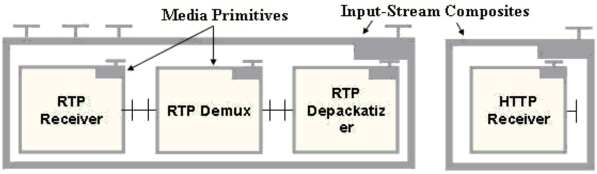

Fig. 2. Examples of Components and Bindings

The advantage of this hierarchy is that it groups primitive components implementing a similar multimedia function under the control of the same composite. This allows the integration of common configuration and reconfiguration operations in the enclosing component independently from the application structure. The combination of various composites constitutes all possible configuration and reconfiguration operations that can be operated on the application. Notice that other levels can be easily added to the component architecture in order to define new composition semantics.

The composition of an application is performed by binding the different components in a flow graph. Both Media composites and Media Primitives participate in this process. Each media primitive exposes one or more stream interfaces used to receive/deliver data from/to other components. A stream interface is typed by the media type of the produced/consumed stream, which is expressed with media-related properties such as MIME type, encoding format and data-specific properties (resolution, colors, etc.). Thus, the success of a binding between two primitive components is governed by the media-type compatibility between the bound interfaces. That is, a binding will fail if there are mismatches between media types of two components. PLASMA provides a binding algorithm which avoids such failures using two kinds of bindings:

- Primitive bindings bind two components agreeing the same media type. This means that media data is streamed directly from input streams to output streams by using method calls between stream interfaces.
- Composite bindings are special composite components which mediate between components handling different media types. Their role is mainly to overcome media type mismatches between MPs. These bindings are made-up of a set of MPs implementing fine-grain media conversions. Figure 3 shows an example of a composite binding between a *Decoder* and an *Encoder* composite. The Decoder provides video data in YUV, while the Encoder accepts only RGB. Moreover, as this later uses H.261 encoding, it only accepts video data in specific resolutions such as QCIF (176*144). The composite binding creates two primitive sub-components: a Resizer to transform the video resolution into QCIF and a YUV-2-RGB to convert data format from YUV to RGB.

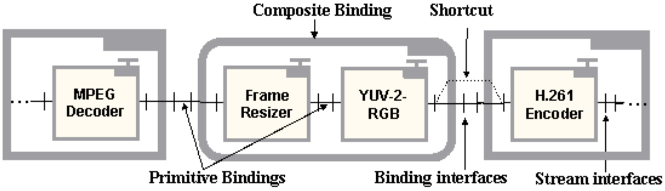

Fig. 3. Examples of component bindings

3.2.2 Probes

Probes define observation points in charge of gathering performance information on both application and system resources. The information is not processed by probes, but only made available to other components wishing access to it. However, they may produce data at different scales and units and thus apply conversions. PLASMA provides two kinds of Probes:

1. *QoS Probes*: Some components are expected to maintain information reflecting QoS values. For example, an RTP Sender component continuously measures packet loss rate, transmission rate, etc. QoS Probes interact with those components to collect QoS information and make it available for other components.
2. *Resource Probes:* They act as separate monitors for gathering resource states such as CPU load, memory consumption, remaining battery life-time, etc.

3.2.3 Sensors

Sensors are used to trigger events likely to activate reconfiguration operations. We distinguish two kinds of Sensors:

1. *QoS and Resource Sensors*: They are generic components associated with QoS and Resource Probes. Their role is to inspect the evolution of observed parameters and to notify relevant changes. The behavior of such Sensors is quite simple: it consists in comparing the observed values with agreed-upon thresholds in order to detect changes in the observed entities. When a change occurs, the Sensor feeds back a corresponding event to the appropriate components.

2. *External-Event Sensors*: They are in charge of monitoring external events requiring a specific implementation. As an example, a Sensor may implement a conferencing manager listening for new connections and notifying the arrival of new participants. A second example would be a Sensor associated with the graphical user interface that sends relevant events.

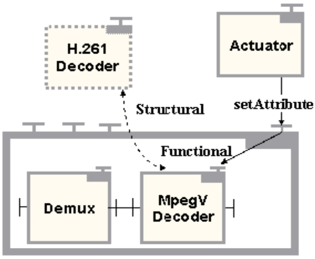

Fig. 4. Examples of reconfigurations

3.2.4 Actuators

Reconfiguration actuators are primitive components responsible for the execution of reconfiguration actions. Actuators react to events by performing required operations on the appropriate components. Each reconfiguration action on a component is performed through its attribute control interface (i.e. by modifying one of the component's attributes). This means that the Actuator has a generic behavior whatever the targeted components. It belongs to the component implementation to decide how to interpretate modifications of its attribute values. We distinguish three kinds of reconfiguration:

- *Functional Reconfigurations*: The most basic form of reconfiguration consists in changing the functional attributes of a primitive component belonging to the application. Reconfigurations may target key attributes in order to tune the produced media stream. As illustrated in Figure 4, the *Decoder* composite may delegate/forward modification of one of its attributes (e.g. 'quality') to its *MPEG-Video Decoder* sub-component, which performs the effective operation (and change the quality of the produced stream). Although such operations only affect media primitives, their impact on the media type agreed during initial bindings of this component defines two cases:
  1. In a first case, the targeted attribute does not affect the media type and therefore, this operation does not interrupt the execution of the application.
  2. In a second case, the targeted attribute affects the media type (e.g. a modification of the video resolution attributes or the representation format). This operation may require unbinding and re-binding the involved components according to rules explained in section 3.2.1.
- *Structural Reconfigurations*: This second form of reconfiguration concerns the modification of a composite's structure, being built-up of a set of sub-components. For instance, a modification of attribute 'encoding-format' of composite *Decoder*

leads to replace the *MPEG-Video* decoder with an appropriate one (*H.261 Decoder* in Figure 4). In general, this operation involves several steps illustrated in Figure 5:

- The *Activation* step includes the detection of changes (Probes/Sensors) and the decision of the execution of reconfiguration operations (Actuators).
- The *Pre-reconfiguration* step encompasses all tasks that can be performed without application interruption, among them: creating new components and setting their initial attributes. Stopping the application may be delayed in order to reduce the application black-out time.
- The *Blackout* step represents the period during which the application is stopped. This is necessary to unbind, remove and/or insert, and bind components.
- The *Post-reconfiguration* step encompasses all tasks that can be made after application restart, among them: deletion of old components and resetting of key attributes.

- *Policy Reconfigurations*: In the third form, reconfiguration actions target reconfiguration components themselves. Some of such reconfigurations are quite similar to functional reconfiguration and involve the modification of key properties of Probes, Sensors and Actuators. Examples consist in changing probing periods, tuning observation thresholds, modifying operations and operand values of reconfiguration actions, etc. Other reconfigurations may target the execution of these components by invalidating reconfiguration actions or activating/deactivating sensors.

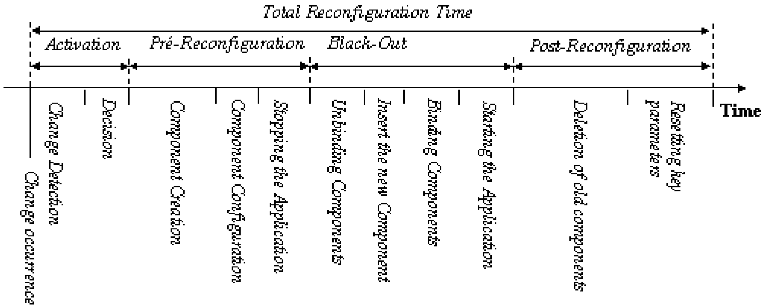

Fig. 5. Execution sequence of structural reconfiguration

4 Use Case: Mobile Multimedia Applications

PLASMA has been entirely implemented in C++ under Microsoft Windows 2000 using the Microsoft .Net 2003 development platform. Multimedia processing and networking functions are based on the Microsoft DirectShow toolbox. Several application scenarios have been successfully released, among them the SPIDER application, a media transcoding gateway for mobile devices.

4.1 SPIDER Architecture

The SPIDER architecture assumes several multimedia sources made available in the network, providing content in various data encoding formats and transmission protocols. Mobile users equipped with PDAs may access these sources; however they are limited to HTTP-based streaming and support exclusively MPEG-1 streams. The role of SPIDER gateways is to mediate between clients and servers by performing appropriate data conversions. A SPIDER node grants access to any multimedia source available in the network by means of data transcoding, transformation, protocol conversion, etc. Media streams are *receiver-driven*, i.e. client applications precise their media preferences (resolution, colors, bit-rate, etc.) as well as adaptation policies to adapt media streams when required.

A typical application scenario is as follows: the client application starts streaming video from a TV broadcast server (originally encoded in H.261 and transmitted using RTP). It requests a transcoding process which converts video content to MPEG with a resolution at 320x240. Knowing that potential increase of the transmission rate may overload its CPU and cause poor QoS (the gateway uses a Variable Bit-Rate encoding), it requests an adaptation policy which decreases the video resolution by 5 % whenever the transmission rate exceeds 512 Kbps. This adaptation algorithm is evaluated periodically (every 10 seconds in our experiment) in order to observe the behavior of the client application during different stages.

```
<TaskFlow id="Server" location="oxygene.inria.fr">
 <Task name="Input-Stream" id="C">
  <Attributes signature="InputAttributeController">
   <Attribute name="src" value="rtp://ozone.inria.fr:5000"/>
  </Attributes>
  <Binding id="b1" client="this" server="E" />
 </Task>
 ....
 <Task name="Video-Encoder" id="E">
  <Attributes signature="EncoderAttributeController">
   <Attribute name="format" value="32" />
   <Attribute name="resolution" value="320x240" unit="pixels" />
  </Attributes>
  <Binding id="b4" client="this" server="D"/>
 </Task>
     ...
 <Task name="Output-Stream" id="O">
     ...
 </Task>
 <Observation id="ob1" type="Resource" resource="id(O)@datarate">
  <event id="evt1" operator="exceeds" value="512" unit="kbps"/>
  <event id="evt2" .../>
 </Observation>
 <Action-set id="set1" condition="evt1">
  <Action operation="decrease" target="id(E)@resolution" operand="5"/>
 </Action-set>
                ….
</TaskFlow>
```

Fig. 6. A Dynamic ADL Description

Such information is conveyed from a client device to a SPIDER node as an ADL description sent using traditional session protocols. In our implementation, ADL descriptions are embedded within HTTP requests. Relying on PLASMA, each SPIDER node translates the ADL description into the suited multimedia adaptation service. Figure 6 shows the ADL description corresponding to the previous scenario.

The application architecture consists of a set of *Tasks* expressing high-level multimedia functions implemented by media composites. Each of them has a collection of attributes that precise its functional properties and its relationships with other Tasks (i.e. media bindings). It results in a task-graph representing the data processing sequence. In our scenario, the application is composed of four Tasks: an Input-Stream, a Video-Decoder a Video-Encoder and an Output-Stream. It receives an original RTP stream (see *src* attribute), decodes its content, encodes the result in MPEG-1, and transmits the result using TCP.

Reconfiguration policies are expressed in terms of *Observations* and *Action-sets*. Observations represent Probes and can be related to Task attributes or to resources (QoS or Resource Probes). Each observation defines one or more events reflecting violations of thresholds associated with observed parameters (i.e. Sensors). Here, a observation is associated with the data rate of the Output stream. Action-sets define one or more actions manipulating attribute values (*target* attribute).

4.2 Performance Evaluation

In our scenario, we used as SPIDER gateway a PC running Windows 2000, equipped with a 2 GHz Pentium 4 processor, 256 MB of Memory and 100 MB/s LAN. On the client side, we used a Compaq IPaq PDA equipped with a XScale Processor at 400 MHz, 64 MB of memory and 11 MB/s 802.11b WLAN. The performance evaluation concerns the cost of reconfiguration operations and their impact on performance of client device.

4.2.1 Performance of Structural Reconfiguration

The first reconfiguration occurrence requires a structural reconfiguration which adds a Video-Resizer component. The whole reconfiguration time is about 36 ms (milli-seconds). The major part is devoted to the pre-reconfiguration step with 24 ms, required for the instantiation of the new component. The blackout time is about 11 ms, spent in order to insert the Resizer component. Post-reconfiguration operations take about 200 s (micro-seconds). We deduce from these results that performing the pre-reconfiguration step before stopping the application significantly minimizes the cost of structural reconfiguration, in our case by 68 % assuming that a naive implementation would have required application be stopped during the whole time.

Subsequent reconfiguration operations on video resolution are only functional reconfigurations on the Resizer components. As these reconfigurations affect the output media type (the resolution), it requires re-binding (unbind, property modification and rebind) of components that are placed after the Resizer component in the processing graph. Setting component attributes takes 67 s. The blackout time is 331 s, where 244 s are necessary to re-bind components. Despite this short blackout, the whole reconfiguration time is low and does not introduce a significant overhead (i.e. about 398 s).

4.2.2 Impact on QoS and Resource Usage

In order to evaluate the impact of reconfiguration on the QoS and resource usage on the client device, we measured the rate of video display (in frames/seconds) observed by the application and the CPU. Figure 7 reports results. Vertical dotted lines mark the occurrences of reconfigurations, detected as a change in the video resolution. In the first time segment, video is displayed at less than 10 fps due to a CPU usage at 100 %. This is due to the fact that application receives a large amount of data and in addition, it must resize video frames in order to fit its display limitations. This operation requires additional processing causing the application to drop frames in response to CPU overload. The first reconfiguration reduces the video resolution to 304x228 and therefore, increases the frame rate to 17 fps. However, the CPU load is kept at 100 % as the transmission rate remains high. Finally, the resolution is reduced to 273x205, resulting in a frame rate at 25 fps and a CPU load at 70 %. These results show on one hand that reconfigurations don't have a negative impact since the reconfiguration blackout time is almost transparent to the client application. On the other hand, they significantly improve QoS and resource usage on the client.

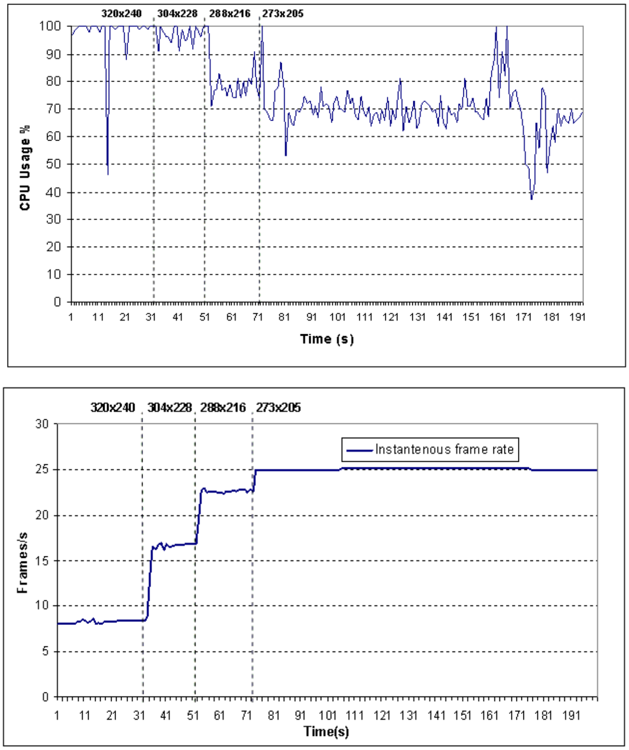

Fig. 7. Impact of reconfiguration on QoS and resource usage

5 Conclusion

This paper has presented PLASMA, a component-based framework for building self-adaptive multimedia applications. To reach this goal, PLASMA relies on an advanced component model whose main feature is the recursive composition of applications. Relying on this feature, a hierarchical reconfiguration management is introduced at different levels of hierarchy. This allows releasing a large scope of reconfiguration independently from the application structure and thus, it offers a flexible approach to adaptation. Our experimental study has shown that component-based adaptation as in PLASMA provides a good trade-off between flexibility and performance. Indeed, it does not introduce a high overhead and can significantly improve QoS and resource usage of multimedia applications.

References

1. G. Blair, L. Blair, V. Issarny, P. Tuma, A. Zarras. The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms. Middleware Conference, April 2000.
2. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma and J-B. Stefani. An Open Component Model and its Support in Java. International Symposium on Component-based Software Engineering, May 2004.
3. A.P. Black, and al. Infopipes: an Abstraction for Multimedia Streaming. In Multimedia Systems. Special issue on Multimedia Middleware, 2002.
4. E. Cecchet, H. Elmeleegy, O. Layaïda and V. Quéma. Implementing Probes for J2EE Cluster Monitoring. In OOPSLA Workshop on Component and Middleware Proformance, Vancouver, October 2004.
5. L.S. Cline, J. Du, B. Keany, K. Lakshman, C.Maciocco, D.M. Putzolu. DirectShow RTP Support for Adaptivity in Networked Multimedia Applications. IEEE International Conference on Multimedia Computing and Systems, 1998.
6. Microsoft: DirectShow Architecture. http://msdn.microsoft.com/directx 2002.
7. D. Duke and I. Herman. A Standard for Mulimtedia Middleware. ACM International Conference on Multimedia. 1998.
8. T. Fitzpatrick and al. Design and Application of TOAST: An Adaptive Distributed Multimedia Middleware. International Workshop on Interactive Distributed Multimedia Systems, 2001.
9. X. Fu and al. CANS: Composable, adaptive network services infrastructure, USITS 2001.
10. H. Schulzrinne and al. RTP: A Transport Protocol for Real-Time Applications, 2003.
11. D. Sisalem and H. Schulzrinne. The loss-delay based adjustment algorithm: A TCP-friendly adaptation scheme. Proc of NOSSDAV '98, July 1998.
12. B. Stiller, C. Class, M. Waldvogel, G. Caronni, and D. Bauer, A Flexible Middleware for Multimedia Communication: Design, Implementation, and Experience," IEEE Journal on Selected Areas in Communications, September 1999.
13. Sun: Java Media Framework API Guide. http://java.sun.com/products/javamedia/jmf/ 2002.
14. L.A. Rowe, Streaming Media Middleware is more than Streaming Media. International Workshop on Multimedia Middleware, October 2001.

15. M. Lohse, M. Repplinger, P. Slusallek, An Open Middleware Architecture for Network-Integrated Multimedia, Joint IDMS/PROMS workhop 2002.
16. K. Nahrstedt, D. Wichadakul, and D. Xu. Distributed QoS Compilation and Runtime Instantiation. IEEE/IFIP International Workshop on QoS 2000.
17. S. McCanne and V. Jacobson. VIC: A flexible framework for packet video. ACM Multimedia Conference, 1995.
18. Z. Morley and al. Network Support for Mobile Multimedia using a Self-adaptive Distributed Proxy, NOSSDAV-2001.
19. D.G.Waddington and G.Coulson, A Distributed Multimedia Component Architecture, 1st International Workshop on Enterprise Distributed Object Computing, October 1997.
20. D. Wichadakul, X. Gu, K. Nahrstedt, A Programming Framework for Quality-Aware Ubiquitous Multimedia Applications, ACM Multimedia 2002.