PLASMA: A Component-based Framework for Building Self-Adaptive Multimedia Applications

Oussama Layaida and Daniel Hagimont

SARDES Project, INRIA Rhône-Alpes ZIRST-655, Avenue de l'Europe- 38334 Montbonnot Saint-Ismier Cedex; France

ABSTRACT

With the proliferation of networked devices, today's multimedia applications operate, as never before, in heterogeneous and dynamic environments. An attractive way to deal with this situation is to make applications self-adaptive (or self-reconfigurable); that is, make them able to observe them-selves and their environment, to detect significant changes and to reconfigure their own behavior in QoS-specific ways. This issue has made the subject of numerous works, especially in the context of multimedia applications. However, several key requirements of adaptivity have not been well addressed such as: the generality to a wide range of applications, the customizability to each application context and the flexibility of reconfiguration mechanisms. We address these aspects in a component-based framework for building self-reconfigurable multimedia applications, named PLASMA. This paper describes the architecture of PLASMA and shows its use through an application use case. Experimental evaluations show that reconfigurations have a low cost, while significantly improving the QoS.

1. INTRODUCTION

Nowadays, multimedia technologies play a central role for many social, entertainment and scientific applications. Many standards have been developed to provide advanced multimedia functionalities such as composite media objects, interactive presentations, 3D animations, etc. This evolution has been accompanied by the apparition of new networked devices with improved multimedia capabilities. This has led, as never before, to heterogeneous and dynamic environments. Networks vary in nature and performance; also, end user devices have diverse computation capabilities in terms of processing power, display, battery lifetime, etc. On the other hand, end-system and network resource availability vary unexpectedly during applications run-time. Such resource fluctuations may violate the requirements of multimedia application and cause a low-level quality of service.

An attractive way of dealing with such changes is to adapt the behavior of applications accordingly. This is achieved through dynamic reconfiguration, the purpose of which is to evolve the initial configuration of the application into a second one that matches the new environment conditions. By this way, applications will be able to optimize their quality of service at any time. The ideal consists in making applications self-adaptive, or self-reconfigurable. That is, allowing them to observe themselves and their environment, to detect significant changes and reconfigure their own behavior in QoS-specific ways. This design principle, called reflection,[16] offers significant advantages in developing self-adaptive systems, middleware frameworks and applications.

1.1. Problem formulation

Bringing adaptivity to multimedia applications is not straightforward. There are many issues that have to be considered in order to provide an accurate solution. From our point of view, the most important are as follows:

- **Generality:** Different applications may have different performance criteria and consequently, they require different reconfiguration strategies. For example, a real-time videoconferencing application is more sensitive to network fluctuations and requires an adapted congestion control algorithm. Such application-specific constraints require the approach to reconfiguration be general enough to be used for a broad range of applications.

Further author information: E-mail: First.Last@inrialpes.fr, Telephone: +33 4 76 61 52 79, Fax: +33 4 76 61 52 52

- **Context-awareness:** Reconfiguration requirements may also vary within the same application depending on its execution context, such as the hardware capacities of the terminal. For instance, a VoD client running on PDA prefers that server reduces the video resolution when congestion occurs; whereas in the case of a desktop PC it would be more efficient to change the encoding format. Such variations of strategies make difficult, nay impossible, for application designers to predict all possible situations at development-time. Therefore, it becomes necessary that reconfiguration policies be easily tailored to the context of each application.

- **Adaptivity:** Some changes in the underlying environment may affect reconfiguration policies. This requires reconfiguration policies be themselves reconfigurable in order to ensure their efficiency during the application life-time.

1.2. This Paper

The primary objective of our work is to deal with the complexity of multimedia applications. Toward this goal, we have designed PLASMA [*], a middleware framework that eases building self-adaptive multimedia applications. We present the framework architecture and show how it addresses the above-mentioned requirements.

The rest of this paper is organized in five sections. The next section presents a classification of related work. Section 3 describes the basic concepts used to design PLASMA. Section 4 details the framework architecture and its main components. Section 5 describes a use case in a video streaming application. Section 6 gives implementation details and some numerical results. Finally, section 7 dresses the conclusion and outlines the future work.

2. RELATED WORK

During the past decade, the development of adaptive systems has been greatly enhanced by middleware and component-based systems.[1] Such component-based systems provide application developers with high-level abstractions that relieve them from dealing with recurrent functionalities, such as remote procedure call, component migration, asynchronous interactions, etc. Following this principle, work around multimedia applications has led to the development of several component-based frameworks such as DirectShow (Microsoft)[8] JMF (Sun)[15] and PREMO (ISO).[9] 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 components. Likewise, reconfiguration operations are facilitated through high-level component-related operations such as: adjusting component's 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. These advantages have motivated several research works. We reviewed the state of the art in this area and found mainly five approaches:

- *Static, hard-coded reconfiguration policies:* A first approach to reconfiguration uses static reconfiguration policies to deal with specific, pre-determined changes in the environment. The MBONE tools VIC[23] (Video Conferencing Tool) and RAT[13] (Robust Audio Tool) are two well-known examples of adaptive applications. Although such applications are not component-based, they use the RTCP[19] feedback mechanism and a loss-based congestion control algorithm in order to dynamically adapt media streams to the available bandwidth. The reconfiguration operation consists in tuning key encoding properties (quality factor, frame rate, etc.) in order to adjust the transmission rate appropriately. Nevertheless, employing static reconfigurations is too restrictive because they have to be anticipated at development-time. Thus, a modification of a reconfiguration policy requires additional development efforts.

- *Component-based frameworks with reconfiguration capabilities:* Some research works have proposed component-based frameworks with reconfiguration capabilities. The Toolkit for Open Adaptive Streaming Technology (TOAST)[10] explores the use of open implementation and reflection to ease the development of adaptive

[*]PLASMA stands for **PLA**tform for **S**elf-adaptive **M**ultimedia **A**pplications

multimedia applications. TOAST offers two kinds of reconfigurations: (1) minor adaptation refers to modifications of component properties and (2) major adaptation involves changes of applications structure by inserting and/or removing components. Although TOAST offers the required features to achieve reconfigurations, they are left to application developers who must deal with: resource and application monitoring, reconfiguration decisions and their implementation.

- *Component-embedded reconfiguration policies:* In some component-based architectures, reconfiguration features are embedded in functional components themselves. In Microsoft DirectShow[8] 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*). QoS control in DirectShow is limited to adjusting the rate of data flow through components to runtime performances. However, it can be easily extended to support a larger scope of QoS control, as proposed in.[6] As in the first approach, reconfiguration operations are hard-coded in components. The major drawback here is that, because components are built at the same level, reconfigurations occur in the context of each component rather than in the application. Then, although it is possible to change the behavior of components, it is not possible to replace a component, as the structure of the application is not known by components.

- *Separate reconfiguration managers:* In contrast 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 applies reconfiguration operations. CANS (Composable Adaptive Network Services)[11] for example, follows this approach to design adaptive media transcoding gateways. A similar work was conducted in[25] to provide adaptive media streaming for mobile devices. The limitation of this approach consists in determining how managers can apply reconfigurations with different kinds of component configurations (corresponding to different multimedia services). Managers must have a perfect knowledge of application structure in order to perform reconfiguration appropriately. Given the high number of possible configurations, this approach remains applicable to a specific class of applications and can not be generalized to a wide range of applications. On the other hand, a change in component implementation or adding new components requires an update of the manager code.

- *Scripting languages for reconfiguration:* Recent works have proposed more flexible reconfiguration schemes, where reconfiguration policies are declared separately from the application using scripting languages as proposed in.[22] Reconfiguration specifications are translated at application launch-time, which facilitates their customization according to application requirements. However, existing works restrict the specification to reconfiguration policies and make hypothesis on applications structure. Indeed, reconfiguration specifications do not designate which components initiate reconfiguration events and which should be targeted by reconfiguration actions. The result is that each application needs a specific translator, which has the knowledge of its structure.

3. DESIGN CONCEPTS

This section presents the main concepts followed for the design PLASMA in order to overcome the shortcomings of previous works. We briefly present the component model used as its basis. We then discuss our approach to reconfiguration.

3.1. Using the Fractal Component Model

We base PLASMA on Fractal[3][†], a software composition framework intended to the deployment, the introspection and the reconfiguration of complex software systems. Compared to other component models, Fractal has the following particularities (see Figure 1):

[†]Fractal is a joint effort of France Telecom R&D and INRIA Rhône-Alpes, and is a project of the ObjectWeb consortium. A detailed specification is provided at http://fractal.objectweb.org

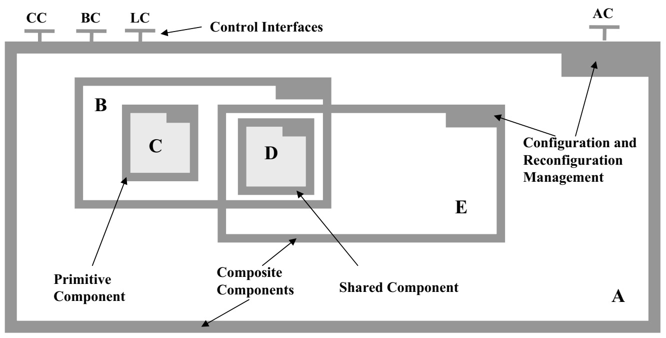

Figure 1. Architecture of enhanced Fractal Components.

- **Hierarchical composition:** The Fractal model defines primitive and composite components. A primitive component encapsulates functional code (C and D on Figure 1) while composite components encapsulate other components that can be either primitive or composite (A, B and E). The model is recursive in the sense that components can appear at arbitrary levels of composition with a similar structure, hence the name Fractal.
- **Modularity:** Fractal components are structured in terms of content and controller. The content part defines a finite number of components, called sub components, which are under the control of the controller of the enclosing component. The controller part embodies the control behavior associated with a particular component. The component model defines four kinds of control interfaces defined as follows:
  - *Binding Controller (BC):* provides operations to bind and unbind interfaces of the component.
  - *Content Controller (CC):* allows listing, adding and removing sub-components in the component content.
  - *Life-cycle Controller (LC):* allows explicit control on the component execution (e.g. start and stop operations).
  - *Attribute Controller (AC):* allows reading and writing the component attributes from its outside.
- **Component sharing:** One characteristic of the Fractal model is the ability to share components between several composite components (component C). Component sharing is very useful to model resource sharing between components.

3.2. Approach to reconfiguration

Hierarchical reconfiguration: Our approach to reconfiguration differs from the previous work in the fact that it employs a hierarchical reconfiguration model. Based on the hierarchical composition provided by Fractal, reconfigurations can also be considered at different levels of composition. As shown on figure 1, each component has its own configuration and reconfiguration manager. By exploiting the knowledge of its structure, each component determines the appropriate way to achieve a given reconfiguration, transparently to higher levels.

Using a Dynamic ADL: The second concept used in PLASMA is the use of a dynamic ADL (Architecture Description Language) within the middleware. Besides structural and functional information of applications, a dynamic ADL offers constructs to describe their dynamic behavior with respect to changes in the environment. The middleware framework offers the required tools to match a specification with a component assembly.

4. THE DESIGN OF PLASMA

This section presents the general architecture of PLASMA. The framework mainly encompasses three kinds of components: *Media Components*, *Monitoring* and *Reconfiguration* components. The following subsections describe the role of each of them, and detail their relationships.

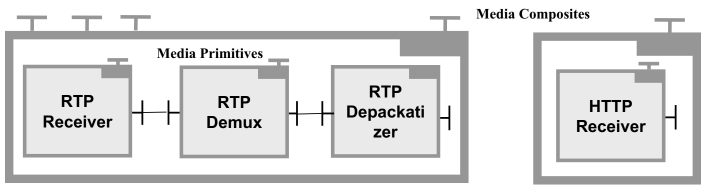

Figure 2. Examples of Media Components: Input-Stream MCs.

4.1. Media Components

Media components represent the computational units used for the composition of the various multimedia services. The architecture is decoupled into three hierarchal levels, each providing a specific functionality:

1. *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. Each media primitive has a set of stream interfaces used to receive/deliver data from/to other components. A stream interface is typed by the media stream formats expressed with media-related properties such as MIME type, encoding format and data-specific properties (resolution, colors, etc.). Thus, each component accepts data in given media formats and produces data in given media formats.

2. *Media Composite (MC)* components are composite components that represent higher-level functions, called *Tasks* , such as Decoding, Encoding, Network Transmission, etc. Each media composite deals with a group of MPs offering various implementations of its Task. It creates, as sub-components, a set of MPs in order to achieve a specific behavior. In the example shown on figure 2, an InputStream composite may be composed of three Media Components 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 role of media composites is to hide all features inherent to their sub-components. They are responsible for creating, configuring and reconfiguring their sub-components when needed. Media composites have one input and one output binding interfaces in order to be bound with other composites. In contrast to stream interfaces, binding interfaces are collection interfaces i.e. they can be bound to several binding interfaces at the same time.

3. The *Media-Session (MS)* component is a composite that 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, reward, etc.).

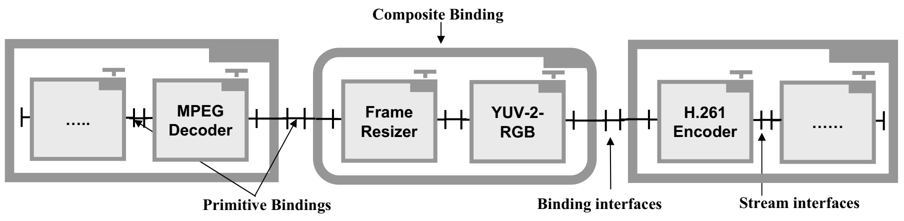

Figure 3. Examples of bindings.

4.2. Media Component Interactions

The construction of an application is performed by binding the different components in a flow graph. Inside each MC, MP components are bound through stream interfaces. Bindings between MCs are performed through their binding interfaces, which map each binding to the appropriate stream interface of their MPs. The success

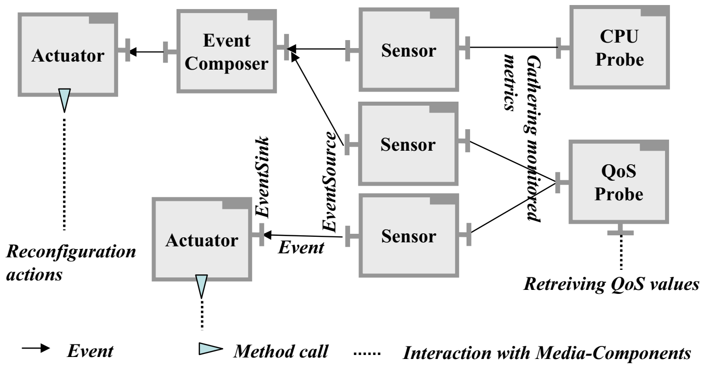

Figure 4. Event-based interactions.

of a binding between two primitive components is governed by the media-type compatibility between the bound interfaces. According to this condition, there may be two kinds of bindings:

- *Primitive bindings* are used to bind components handling 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 that mediate between components handling different media types. Their role is mainly to overcome media type mismatches. 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 composites. The Decoder composite provides video data in YUV, where Encoder composite accepts only RGB. Moreover, as the later uses H.261 encoding, it 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.

4.3. Monitoring and Reconfiguration Components

In addition to media components, our framework defines components to coordinate reconfiguration operations within applications. These components can be inserted at any level of composition, i.e. as sub-components of Media Composites or of the Media-Session component. Figure 4 shows a possible assembly of these components. We distinguish three components added for this purpose: Probes, Sensors and Actuators.

4.3.1. Probes:

Probes define observation points that can be inserted at any level of the composition. They implement the required operations in order to gather resource or application states. We distinguish two kinds of Probes:

- *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.
- *Resource Probes*: Resource Probes act as separate monitors gathering resource states such as CPU load, memory consumption, remaining battery life-time, etc. Our framework provides a set of resource Probes that can be configured in order to return various metrics of each resource.

4.3.2. Sensors:

The role of Sensors is to trigger events likely to activate reconfiguration operations. We distinguish two kinds of Sensors:

- *QoS and Resource Sensors :* The first kind of sensors is associated with QoS and Resource Probes. Their role is to notify relevant changes of observed parameters. The behavior of QoS and Resource Sensors is generic: it consists in comparing the observed values with agreed thresholds in order to detect changes in the observed entity. When a change occurs, the Sensor feeds back a corresponding event to the appropriate components.
- *External-event Sensors :* The second kind of sensors monitors external events. These Sensors may be used for different purposes, each 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.

This separation into Probes and Sensors brings significant flexibility to achieve monitoring functionalities. Indeed, several events can be issued from the same observation. Also, basic events can be composed in order to trigger composite events before actuating reconfigurations. Such features can be accomplished through Probes and Sensors, as described on Figure 4.

4.3.3. 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 AttributeController (AC) interface. This interface provides the following methods for attributes control:

- `hasAttribute(string name)`: queries the component on the presence of a given attribute.
- `string getAttribute(string name):` retrieves the value of an attribute given its name.
- `setAttribute(string name, string value, string unit)` sets the value of an attribute.

The behavior of the Actuator is the same for each kind of components. Indeed, all reconfigurations are executed as changes of component attributes, and achieved through the AC interface. It belongs to the component implementation of this interface to decide how to execute modifications of its attributes' values. Depending on the targeted component and the semantic associated to its attribute; a reconfiguration may lead to one of the following operations:

1. *Functional reconfigurations:* The most basic form of reconfiguration consists in changing the functional attributes of a primitive component belonging to the application. Based on this, reconfigurations may target key attributes in order to tune the media stream to the desired state. For example, changing the encoding quality factor or the frame rate of the encoder component.
2. *Structural reconfigurations:* This operation may occur in the context of the composite, being built-up of a set of sub-components. It involves: adding/removing a sub-component, changing component bindings, stopping/starting sub-components.
3. *Policies reconfigurations:* Reconfiguration actions may also target reconfiguration policies themselves. Some of them are similar to parameter reconfiguration and requires the modification of key properties of Probes, Sensors and Actuators. Examples in that sense may consist in changing probing periods, tuning observation thresholds, modifying operations and operand values of reconfiguration actions, etc. Other reconfigurations target the execution of these components by invalidating reconfiguration actions, stopping/restarting probes and sensors.

4.3.4. Event-based Interactions:

The communication between Probes, Sensors and Actuators follows a simple event-based model based on three interfaces: Event, EventSource and EventSink. Event is used to identify any event, defined with specific properties such as event id, occurrence date, duration, priority, source and specific data. The EventSource interface is implemented by event emitters (e.g. Probes and Sensors). This interface provides methods that allow: (1) querying supported events, (2) registration for events, and (3) deregistration from events. The EventSink interface is supplied by event receivers (e.g. Actuator) in order to be notified of event occurrences.

Other components are derived from these interfaces mainly for event-based communication issues. An Event-Composer for example (see Figure4) is a component implementing both EventSink and EventSource interfaces. As its name implies, it used for the composition of basic events with boolean constructs. A second example is an Event-Filter component which receives events, and forwards valid ones according to their priorities.

5. USE CASE: SELF-ADAPTIVE VIDEO STREAMING APPLICATION

This section presents a simple use case which consists in building adaptive video streaming application. It shows how PLASMA eases to the design of such applications, and in particular, how reconfiguration capabilities are added to this applications.

```
<TaskFlow id="Server" location="194.199.25.52">
    <Task name="Input-Stream" id="C">
        <Attributes signature="CaptureAttributeController">
            <Attribute name="src" value="camera" />
            <Attribute name="height" value="288" unit="pixels" />
            <Attribute name="height" value="352" unit="pixels" />
        </Attributes>
    </Task>
    <Task name="Video-Encoder" id="E">
        <Attributes signature="EncoderAttributeController">
            <Attribute name="fmt" value="31" />
            <Attribute name="quality" value="80" unit="%" />
        </Attributes>
        <Binding id="b5" client="C" server="this" />
    </Task>
    <Task name="Output-Stream" id="O">
        <Attributes signature="OutputAttributeController">
            <Attribute name="src" value="rtp://194.199.25.99:5004" />
            <Attribute name="packet-loss" value="0" />
            <Attribute name="FEC" value="none" />
        </Attributes>
        <Binding client="E" server="this" />
    </Task>
    <Observation id="obs1" type="Resource" source="RTT">
        <event id="evt1" operator="falls" value="100" unit="%" />
        <event id="evt2" operator="exceeds" value="200" unit="%" />
    </Observation>
    <Observation id="obs2" type="Qos" source="Server/id(C)@packet-loss">
        <event id="evt1" operator="exceeds" value="25" unit="%" />
    </Observation>
    <Action-set id="set1" predicate="evt1">
        <Action operation="increase" target="id(E)@quality" operand="5" unit="%" status="active"/>
    </Action-set >
    <Action-set id="set2" predicate="evt2">
        <Action operation="decrease" target="id(E)@quality" operand="5" unit="%" status="active"/>
    </Action-set >
    <Action-set id="set3" predicate="evt3">
        <Action id="act2" operation="set" target="Server/id(E)@FEC" operand="Reed-Solomon" status="active"/>
        <Action id="act3" operation="set" target="Server/id(set3)/id(act2)@status" operand="passive" status="active"/>
    </Action-set >
</TaskFlow>
```

Figure 5. An ADL Description of a Video Streaming Server.

5.1. Application

The video streaming application consists in two parts: a server continuously distributing an RTP H.261 video content and a client able to display RTP H.261 video streams. To deal with network resource fluctuations, the server adjusts the video quality according to the RTT (Round Trip Time) between client and sever and packet loss ratio observed by the client applications (we employ the RTCP feedback mechanisms to raise packet-loss to the server). The algorithm is quite simple and has three rules: (i) if the RTT ratio falls 100 milli-seconds (ms) then increase the encoding quality by 5 %, (ii) if it exceeds 200 ms then decrease the encoding quality by 5 % and (iii) if packet-loss ratio exceeds 25 % then apply a Reed Solomon FEC (Forward Error Correction) algorithm on the data stream. In the following, we focus on the server part and details how it is built.

5.2. ADL Description

In order to build this server, we use the description on Figure 5. As we can see, a description consists of a set of *Tasks*. Each task may have a collection of attributes that precise its functional properties and its relationships with other Tasks (i.e. bindings). The result of all bindings consists in a task-graph representing the data processing sequence. The server description consists in three Tasks: an Input-Stream, a Video-Encoder and an Output-Stream.

Reconfiguration policies are expressed in terms of *Observation*s and *Action-set*s. Observations can be related to Task attributes (QoS Observations or to resources (*type* attribute). They define events reflecting violations of thresholds associated to observed parameter. Our example defines a QoS observation related to the packet-loss of the Output-Stream Task (see *type* attribute). A second observation defines events related to the RTT measurement. Action-sets define one or more actions manipulating attribute values. For example, the action in the first action-set consists in *decresing* the quality factor attribute (*target* attribute) by 5 %.

5.3. Deployment and Configuration

The translation of the previous description results to the component architecture shown on Figure 6. The application is represented by a MS component composed of:

- Three MCs are created for each Task in the description.
- Two Probes created for each Observation. Also, a Sensor component is created to trigger each of its events when required.
- Three Actuators are created for each Action-set; they are responsible for the execution of actions.

Each of these components is configured with its description values, based on which it will determine its accurate behavior. Each MC, creates one or more MPs according to its functional attributes. The Video-Encoder for example, creates an H261Encoder as is has a *format* attribute set to *H.261*. In a similar manner, each component use ADL information in order to determine its interactions with other ones, either through media bindings or event-based interactions.

5.4. Reconfiguration

Once this application launched, it is subject to reconfigurations that may result of the previously mentioned rules. Our example has three Actuators likely to apply reconfiguration operations. As explained before, such operations depend on the semantic of the targeted attribute. The first and the second reconfigurations consist in tuning the quality factor attribute of the Video-Encoder MC. This later requires: (1) retrieving the current quality value through getAttribute method and (2) setting the new value through setAttribute method. These calls are delegated to the H261 encoder, which modifies its behavior accordingly. The third reconfiguration targets the FEC attribute of the Video-Encoder and consists of two operations:

- The first consists in setting the FEC attribute to Reed Solomon. As shown on Figure 6, this involves the insertion of a new component inside the Video-Encoder (FEC-Encoder with dotted border). This is achieved by: (1) creating the FEC-Encoder component (2) stopping media components and unbinding the H.261-Encoder (3) adding the FEC-Encoder and binding components and (4) restarting media components.
- The second action consists in stopping the Actuator to which its belongs, as these operations can no longer be applied (the FEC encoder has already been added).

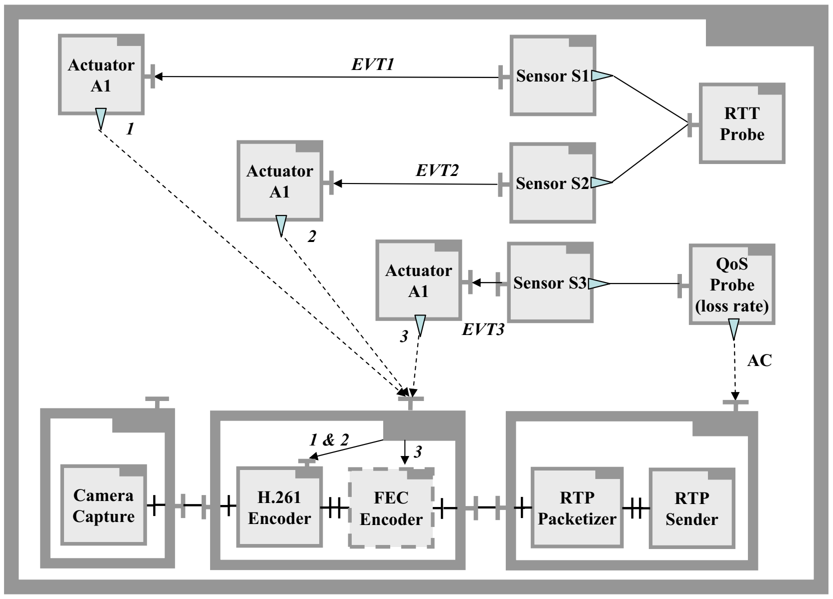

Figure 6. Structure of the video streaming server.

6. IMPLEMENTATION AND PERFORMANCE EVALUATION

The framework has been entirely implemented in C++ under Microsoft Windows 2000 using Microsoft .Net 2003 development platform. Our implementation provides a complete and compliant C++ implementation of the Fractal model, of which the reference implementation is provided only in Java. Multimedia processing functionalities have been developed using DirectX Software Development Kit (SDK) version 9.0. The DirectShow model, a part of DirectX, provides a component library providing basic multimedia functions. Some missing components have been added to implement communication and transformation functions. Resource monitoring features have been implemented in the context of LeWYS [‡], a general purpose framework for monitoring in Linux and Windows platforms. It provides a set of library routines that simplifies the collection of performance data. Windows monitoring routines are implanted upon the Performance interface of the windows registry, which makes them powerful and less costly. Several application scenarios have been built, such as: video transcoding gateways, audio mixing servers, VoD servers, etc. We give below some numerical results obtained in these experiments.

The first evaluation [§] concerns the instantiation cost, which is evaluated as the time spent to parse an ADL description, check its correctness, create all required components and finally launch the application. We have measured this time by varying the number of components through different application scenarios. The obtained results have shown that this time vary strongly with the number of primitive components, which was about 5 ms for each component. The reason is that in DirectShow, media processing components are hosted in DLLs (Dynamic Link Library) which are dynamically loaded for each component creation.

The second evaluation concerns reconfiguration, where we evaluated the time required to replace a component at run-time. This operation has taken about 15 ms, during which the application was stopped. Our previous study[21] in the context of self-reconfigurable proxies, have shown the benefits of such reconfigurations on the QoS perceived by users.

[‡]LeWYS stands for LeWYS is Watching Your System, it is freely available at http://lewys.objectweb.org

[§]We used a Windows 2000-based PC with a Pentium 4 Processor at 2 Ghz and 256 Mo of memory.

7. CONCLUSION

This paper has presented PLASMA, a component-based framework for building self-adaptive multimedia applications. To address the shortcomings of existing works, PLASMA employs concepts of hierarchical reconfiguration and dynamic ADL in order to efficiently bring adaptivity to multimedia applications. The architecture has been described and illustrated through an application scenario. This example has shown how PLASMA eases the development of adaptive multimedia applications, and how reconfigurations are achieved within applications. Our experimental study has shown that employing a component-based design does not introduce a high overhead. Various application scenarios have been released, such as: video transcoding gateways, audio mixing servers, etc.

Our future work will include distribution, where distributed applications may cooperate in order to achieve reconfigurations. This may concern either monitoring features or the execution of reconfiguration operations. With a distributed monitoring, applications can remotely gather resource and QoS information in order to have a fine analysis of application and system states. Having distributed reconfiguration operations means that application may explicitly trigger reconfiguration actions on remote nodes. This allows, for example, the client to act on the server's configuration when needed.

Acknowledgements

This work has been partly funded by the European IST Ozone project (IST-2000-30026) and by a Microsoft Research Innovation Excellence Award.

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. In Proceedings of Middleware 2000, April 2000, Hudson River Valley (NY), USA.
2. G. Blair and J. Stefani: Open Distributed Processing and Mulitimedia. Addison-Wesley 1998.
3. E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and Dynamic Software Composition with Sharing. Seventh International Workshop on Component-Oriented Programming, (WCOP02), Spain, June 10, 2002.
4. A.P. Black, and al. Infopipes: an Abstraction for Multimedia Streaming. In Multimedia Systems. Special issue on Multimedia Middleware, 2002.
5. G. Coulson, G.S. Blair, M. Clarke, N. Parlavantzas, The Design of a Configurable and Reconfigurable Middleware Platform. The Journal of Distributed Computing, 2001.
6. L.S. Cline, J. Du, B. Keany, K. Lakshman, C.Maciocco, D.M. Putzolu. DirectShow(tm) RTP Support for Adaptivity in Networked Multimedia Applications. IEEE International Conference on Multimedia Computing and Systems June 28 - July 01, 1998 Austin, Texas.
7. H. Djenidi, A. Ramdane-Cherif, C. Tadj and N. Levy. Dynamic Based Agent Reconfiguration of Multimedia Multimodal Architecture. In MSE 2002, Fourth International Symposium on Multimedia Software Engineering. Newport Beach, California, USA, 11-13, December, 2002.
8. Microsoft: DirectShow Architecture. http://msdn.microsoft.com/directx 2002.
9. D. Duke and I. Herman. A Standard for Mulimtedia Middleware. In: ACM International Conference on Multimedia. (1998)
10. T. Fitzpatrick and J. Gallop and G. Blair and C. Cooper and G. Coulson and D. Duce and I. Johnson, Design and Application of TOAST: An Adaptive Distributed Multimedia Middleware. In: International Workshop on Interactive Distributed Multimedia Systems, 2001.
11. X. Fu and al. CANS: Composable, adaptive network services infrastructure, USITS 2001.
12. E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley. 416 pp.
13. V. Hardman and al. Reliable Audio for Use over the Internet, INET'95.
14. M. Lohse, M. Repplinger and P. Slusallek, An Open Middleware Architecture for Network-Integrated Multimedia in Protocols and Systems for Interactive Distributed Multimedia Systems, Proceedings of IDMS/PROMS'2002, Portugal, November 26th-29th, 2002.

15. Sun: Java Media Framework API Guide. http://java.sun.com/products/javamedia/ jmf/ 2002.
16. B.C. Smith, "Reflection and Semantics in a Procedural Programming Language", PhD Thesis, MIT, January 1982.
17. H. Schulzrinne and al. RFC-3550 RTP: A Transport Protocol for Real-Time Applications, 2003.
18. L.A. Rowe, Streaming Media Middleware is more than Streaming Media International Workshop on Multimedia Middleware, October 2001.
19. H. Schulzrinne and al. RTP: A Transport Protocol for Real-Time Applications. RFC 1889.
20. H. Schulzrinne. RTP Profile for Audio and Video Conferences with Minimal Control. RFC 1890.
21. O. Layaida, S. Ben Atallah, D. Hagimont. Adaptive Media Streaming Using Self-reconfigurable Proxies. In Proceedings of the 7th IEEE International Conference on High Speed Networks and Multimedia Communications (HSNMC'04), Toulouse, France, June 30-July 02, 2004.
22. B. Li and K. Nahrstedt, A Control-based Middleware Framework for Quality of Service Adaptations, IEEE JSAC, 17(9), 1999.
23. S. McCanne and V. Jacobson. VIC: A flexible framework for packet video. Proc. of ACM Multimedia'95, November 1995.
24. S. McCanne and al. Toward a common infrastructure for multimedianetworking middleware. In Proc. 7th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '97), St. Louis, Missouri, May 1997.
25. Z. Morley Mao and al. Network Support for Mobile Multimedia using a Self-adaptive Distributed Proxy, NOSSDAV-2001.
26. D.G.Waddington and G.Coulson, A Distributed Multimedia Component Architecture, In Proceedings of the 1st International Workshop on Enterprise Distributed Object Computing, Australia, October 1997.