

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computing Science

MASTER THESIS

Advanced Connection Management

by

Ling-Zhen Wu

Supervisors: Dr.ir. P.H.F.M. Verhoeven
Dr.ir. J.J. Lukkien
Dr.ir. A.A.J. de Lange

June 28, 2004

A. ABSTRACT

Consumer electronics (CE) devices nowadays are highly complex, diverse and include many advanced features. It is important that the development of such devices occur within a reasonable amount of time as CE devices become outdated very quickly. The Advanced Connection Management (ACM) defines an architecture in which media processing systems can be easily and rapidly developed, build and prototyped. Herein media processing systems are built from abstract notions called logical components that provide an abstract, focused set of media processing functionality. Implementations of these logical components run distributed on different machines in a network as independent software applications. A distinction is made between the abstract logical component and its implementation to enable reuse and replaceability. For the assessment of product concepts it is essential that different compositions of logical components can be setup and evaluated in a rapid manner such that an impression can be obtained of the usefulness and feasibility of it. The ACM provides a high-level abstraction to easily setup different compositions of logical components

Media processing systems often implement different configurations. Each of these configurations requires a different composition of logical components. Switching of one configuration to another requires a reconfiguration of the connections between the logical components. A key requirement for the reconfiguration process in the media processing domain is that the switch occurs in a reasonable period and that the transition occurs “smoothly”. A technique is described in this thesis to perform this reconfiguration dynamically such that the reconfiguration time is minimized and the transition between configurations occurs “smoothly” and correct.

B. TABLE OF CONTENTS

A.	ABSTRACT	3
B.	TABLE OF CONTENTS	5
C.	ABBREVIATIONS, DEFINITIONS	7
1	INTRODUCTION	9
1.1	Scope	9
1.2	Goal	9
1.3	Research Question	9
1.4	Leading Example	10
1.5	Related work	11
1.6	Outline	12
2	DEVELOPMENT AND PROTOTYPING OF APPLICATIONS	13
2.1	Product Development	13
2.1.1	Development phases	13
2.1.2	Component-based software development	14
2.2	Product Concept Assessment	15
2.2.1	About the assessment of concepts	15
2.2.2	Features and technologies	15
2.2.3	Dynamic Composition	16
2.2.4	Requirements for compositional approach	16
2.3	STAGE Prototyping	17
2.3.1	Focus of STAGE	17
2.3.2	System Architecture	17
3	ADVANCED CONNECTION MANAGEMENT	21
3.1	Introduction	21
3.1.1	Goal	21
3.2	Logical Component	22
3.2.1	Communication with outside world	23
3.2.2	Task	23
3.3	Use Case	24
3.3.1	Use case description	25
3.4	Connection Manager	26
3.4.1	Separation into logical and physical	26
3.4.2	Logical Connection Manager (LCM)	28
3.4.3	Physical Connection Manager (PCM)	29
3.5	Dynamic Plug-in Architecture	31
3.5.1	Self-Describing Logical Components	31
3.6	Dynamic Connection Management	31
3.6.1	Dynamic Interconnection	31
3.6.2	Mapping from logical to physical	33
3.7	Model of computation	34
3.7.1	Kahn Process Network	34
3.7.2	Streaming Model	34
3.8	Universal Plug-and-Play (UPnP)	35
3.8.1	UPnP Architecture Overview	36
3.8.2	Simple UPnP Framework	36

3.8.3	UPnP and ACM	37
3.8.4	Proxy / Stub mechanism	38
4	Use case switching	39
4.1	Switching conditions	39
4.2	Dynamic reconfiguration	39
4.2.1	Reconfiguration techniques	40
4.2.2	Requirements	40
4.3	Dynamic Reconfiguration Strategy	41
4.3.1	Minimization of reconfiguration time	41
4.3.2	Residue Streams	43
4.3.3	Conclusion	44
4.4	Reconfiguration and Half Channels	45
5	RESULTS	47
5.1	Proof of concept	47
5.1.1	Setup of demonstrator	47
5.2	Implementation details	48
5.2.1	Graphical User Interface	48
5.2.2	Media processing system development	50
5.3	Results	51
5.3.1	Plugging of new logical components	51
5.3.2	Reservation policy	51
5.3.3	Flexible software component composition	52
5.3.4	Evaluation use case switching	53
6	EVALUATION	55
6.1	Conclusion	55
6.1.1	Development and building of applications	55
6.1.2	Dynamic reconfiguration	55
6.1.3	Using UPnP technology	56
6.2	Outlook	56
6.3	Future Work	57
6.3.1	Error Handling	57
6.3.2	Design Environment	57
6.3.3	Quality-of-Service (QoS)	58
7	REFERENCES	59
8	APPENDICES	63
8.1	Use Case XML Description File	63
8.2	Logical Model	65
8.3	Sequence Diagrams	66
8.3.1	Use Case Setup	66
8.4	Creation of connection	67
8.5	Teardown of connection	68
8.6	Task Interaction	69
8.6.1	Half-Channel Specification	69

C. ABBREVIATIONS, DEFINITIONS

Abbreviations

ACID	Atomicity, Consistency, Isolation, Durable
ACM	Advanced Connection Management
CE	Consumer Electronics
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
LCM	Logical Connection Manager
PCM	Physical Connection Manager
QoS	Quality of Service
SOA	Service Oriented Architecture
SSA	Storage Systems and Applications
STAGE	SysTEM Architecture for Generic storage Engines
UPnP	Universal Plug-and-Play
XML	eXtended Markup Language

Definitions

Advanced Connection Management	Architecture consisting of a high-level platform abstraction that allows for the installation of components and component frameworks.
Channel	Elegant mean to model communication. Abstraction from the actual communication and streaming mechanisms.
Component Model	Set of standards for component implementation, naming, interoperability, customization, composition, evolution and deployment.
Component Framework	Dedicated and focused architecture that defines the mechanisms and policies that enables a dynamic way for building software component compositions.
Logical Component	Grouping of signal processing functions into logical streaming units that provide standardized interfaces for control and clear linking points for streaming communication.
Signal process function	Low-level process that performs transformation on streams.

Software Architecture	High-level abstraction of a software system that defines a set of components and connections between the components such that certain constraints are met.
Software Component	Independent unit of software deployment that satisfies a set of behavior rules and implement well-defined interfaces.
Software Component Composition	Set of components and connections that define the composition of the components.
Task	Encapsulation of a signal processing function to provide standardized communication mechanisms.

1 INTRODUCTION

1.1 Scope

This report concludes a ten month project that has been performed at the Storage Systems and Applications (SSA) group at Philips Research Laboratories Eindhoven. The SSA group focuses on the storage of audio and video content and the development of smart solutions for the retrieval of stored content. As well as a research report for Philips this report also serves as a master thesis for the System Architecture and Networks group at the Eindhoven University of Technology (TU/e).

1.2 Goal

Due to the ever-increasing complexity and diversity of consumer electronic systems and the shorter time-to-market of new products, various component technologies have been developed to enable the practical (re)use of pre developed software parts that is necessary for coping with these problems. Within the SSA group the STAGE prototyping platform is being developed that enables the quick assessment of product concepts and the fast prototyping of complex architectures. The STAGE platform is targeted at prototyping media processing systems in a distributed manner and is a component-based architecture in which the software is built from clearly separated components to allow for reuse and modularity. The component architecture specifies the set of interfaces and the rules to which the components have to adhere.

An important concept within the STAGE platform is connection management. Connection management is essential for the assessment of product concepts and vital for systems that implement different configurations as it enables an easy way for recombining different components at run-time. The connection management must be able to create logical streaming connections between different components and change the connections between components dynamically.

1.3 Research Question

Using software components for building software systems is an emerging approach as software components enable the practical reuse of software parts. Software components are independent units of software deployment that satisfy a set of behavior rules and implement well-defined interfaces. Developing and building software systems from components that are possibly externally acquired can improve the quality and development time of a software system. An important requirement for building software systems from software components is to have an adequate component architecture and component model that define to what interfaces and rules the components have to conform to, how components can be composed together and how they interoperate with each other.

The goal of the ACM is to create an environment in which media processing systems can be rapidly developed, build and prototyped by adopting a component based approach. For the assessment of product concepts it is essential that the components can be easily recombined without much effort. Applications of distributed media processing systems often operate in mission-critical domains that impose strict

performance requirements. For this reason it is important that the high-level abstraction of the ACM does not go the expense of performance.

In this thesis it is further investigated and further elaborated on the requirements of the ACM and to what extent a component-based development approach for developing media processing systems is feasible. The central research question of this paper is to come up with a design of the architecture for the ACM that needs to suffice the following requirements:

- The ACM provides a dynamical way for easily setting up different software component compositions. The ability to setup different software component compositions is crucial for the assessment of product concepts.
- To enable the development and prototyping of distributed media processing systems the ACM must admit an efficient implementation. Here efficient means that distributed media processing systems must be able to be prototyped in real-time on real data sets to evaluate the usefulness and feasibility of the prototyped media processing system.
- Media processing systems usually implement different configurations. The transition from one configuration into another must proceed “smoothly” and in a reasonable amount of time. Smooth means that the reconfiguration of the software component composition results in minimized perceivability of the reconfiguration by the user in the output stream.
- The high-level abstraction provided by the ACM must not contain any references to the interconnection network of the underlying physical platform. Furthermore it is also important that the ACM makes a clear distinction between the abstract set of functionality provided by the software component and its implementation. This is necessary to allow for portability to embedded systems or other platforms.

An implementation of the ACM is made and it is tested if the implementation meets the described requirements. Furthermore it is evaluated if the ACM proves to be able to tackle the product concept assessment and (real-time) prototyping of distributed media processing systems as described in the previous paragraph. To answer this question a demonstrator is built to prove the concept. This demonstrator is described in more detail in [5.1 Proof of concept].

1.4 Leading Example

To clarify the notions and concepts presented in this paper in a systematic way a leading example is introduced. Consider a security monitoring system of an airport. Thousands of people pass through the airport on a daily basis. For the airports and the people’s safety security cameras are used to track each movement of the people at the airport. The video streams captured by these cameras all come together in a central control center where each stream is recorded into a central audio video database. Because the number of security cameras exceeds the number of security guards only a number of security streams can be monitored by these guards at the same time. The

security monitoring system automatically runs through the security cameras by reconnecting the screen to other security cameras. The possibility also exists for reviewing a stream that has previously been recorded to the audio video database.

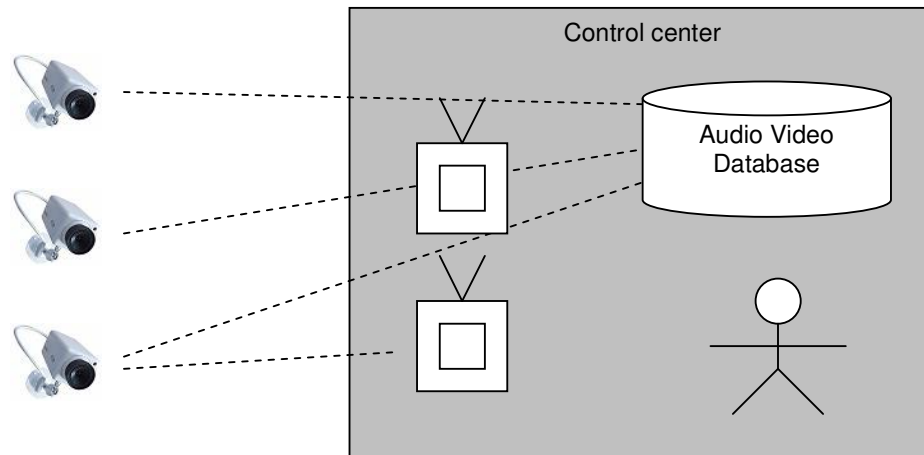


Figure 1: Leading example: security cameras

The different configurations of the security monitoring system described above consist of the display of the video stream on the different screens. It is apparent that certain timing constraints exist; the delay between the capturing of the video stream and the display of it on a screen must be reasonable and furthermore it is important that no streams disappear.

1.5 Related work

GStreamer [GStreamer] is a framework for creating streaming media applications. The development framework of GStreamer has been designed to make it easy to write any type of streaming multimedia application. The framework is based on plug-ins that provides various codec and other functionality. The plug-ins can be linked and arranged together in a pipeline fashion. The pipeline defines the flow of data. Pipelines can be edited in a GUI editor and saved to create pipeline libraries. Currently GStreamer does not provide any support for the dynamical reconfiguration of the pipeline, although in future this functionality will be supported.

[Mitchell et al, 1998] describes the reconfiguration architecture of the DJIN multimedia-programming framework that is designed to support the construction and dynamic reconfiguration of distributed multimedia applications. DJIN applications are constructed from networks of active components consuming, producing and transforming media data streams that are interconnected through their ports. The reconfiguration architecture mainly concentrates on the calculation of schedules for updating peer components that processes media data streams, in order to achieve a glitch-free and timely reconfiguration of multimedia applications. To this end it introduces multimedia transactions that take the applications from one configuration to another while maintaining their temporal, structural and data consistency.

The CINEMA system described in [Rothermel et al, 1994] is a middleware layer that provides a high-level abstraction of the operating systems, services and protocols that

is needed for the efficient development of distributed multimedia applications. It provides a flexible mechanism for dynamic reconfiguration that allows for the definition of arbitrary complex flow graphs connecting various types of multimedia processing elements. An abstraction of a clock hierarchy is proposed to permit grouping, controlling and synchronization of media streams.

In [Ledeczi et al, 2000] self-adaptive systems are modeled in a generic manner to represent dynamic software architectures. In this approach the components of the architecture are prepared, but their number and connectivity pattern are not fully defined at design time. Instead an algorithmic description and architectural parameters are provided along with some generators that reconfigure the system to the described architecture.

1.6 Outline

Chapter 2 gives a brief description of how software development and prototyping traditionally occurs. The different phases in this development process are identified. Next it is argued that the traditional software development process is not adequate for the product concept assessment problems addressed in this paper and explains how the development process is envisioned in the STAGE platform.

In chapter 3 the ACM is explained by first describing the entities involved in the ACM and subsequently describe the architecture of the ACM. Furthermore it is explained how the Universal Plug-and-Play (UPnP) architecture has been incorporated into the ACM as an abstraction layer of the physical network.

Chapter 4 describes a method for switching between use cases. Use case switching is vital for CE devices that implement different configurations. Use case switching among other things entails the calculation of schedules for the dynamic reconfiguration of the underlying interconnections between the different components to ensure a “smooth” and correct transition from one configuration of a media processing system to another configuration.

A proof of concept is presented in chapter 5 of the ACM strategy by means of a demonstrator. An evaluation of the demonstrator is given.

Finally a conclusion is given in chapter 6 and the possible future work is presented.

2 DEVELOPMENT AND PROTOTYPING OF APPLICATIONS

The skeleton and foundation of a software system is its architecture. Software architecture is defined as a high-level abstraction of a software system that describes the functional requirements and the non-functional quality attributes, which the software system has to meet. A software architecture describes the externally visible, overall structure of a system in terms of its components, subsystems and their interconnections and defines the set of basic interfaces and interactions between the components. The architecture is the primal place to address extra-functional requirements. [BASS et al, 1998] defines software architecture as the set consisting of components, connectors and constraints. Connectors between components manage the data or control flow of a system and the constraints define the system's behavior. Designing a software system does not merely consist of designing the software architecture, but also needs to define a design process that describes the steps that need to be followed for connecting the components together such that they meet the specified constraints.

This chapter describes the steps that are traditionally followed in software development and explains how a design process traditionally proceeds and then moves on to describing that the traditional design and development process is not adequate for tackling the problem of product concept assessment and prototyping of distributed media processing systems. After this the requirements for an adequate design process are identified. Finally an overview of the STAGE platform is given that is being designed to address these problems.

2.1 Product Development

2.1.1 Development phases

The product development process can traditionally be subdivided into three phases: analysis, design and implementation.

In the analysis phase software developers determine the precise requirements of the product under development. The requirements are commonly determined by identifying the user's needs through interviews or questionnaires. Each requirement must be clear and testable. This phase is mainly targeted at the user's problem.

After the requirements of the product have been established, the software developer thinks about how these requirements can be realized. In the design phase the software developer turns the requirements into a more specific and detailed product specification and a design model is made. The design model consists of a "blue print" of the software architecture and roughly identifies the building blocks of which the software architecture consists of. Building blocks can consist of functions, objects or even components and it is important that the design model is independent of any technology or programming language in order to be flexible. Important criteria for the design model are extensibility, correctness, reusability and robustness.

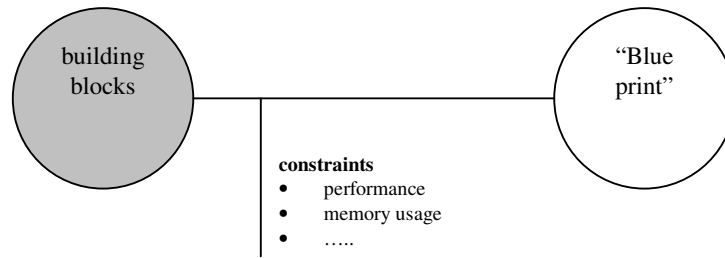


Figure 2: Design model of software architecture

Once the design model has been created and thoroughly analyzed, the software developer can think of which technology he is going to use and which programming language is most suitable for implementing and realizing the system. He has to determine how the different building blocks can be “glued” together according to the blue print such that it meets the constraints set by the design model.

2.1.2 Component-based software development

A commonly used approach for dealing with the complexity of developing software systems is to adopt a “divide and conquer” strategy in which the complex problem is divided into smaller problems that are easier to grasp and handle. Figure 3 shows the paths that are usually followed when developing a software system from components.

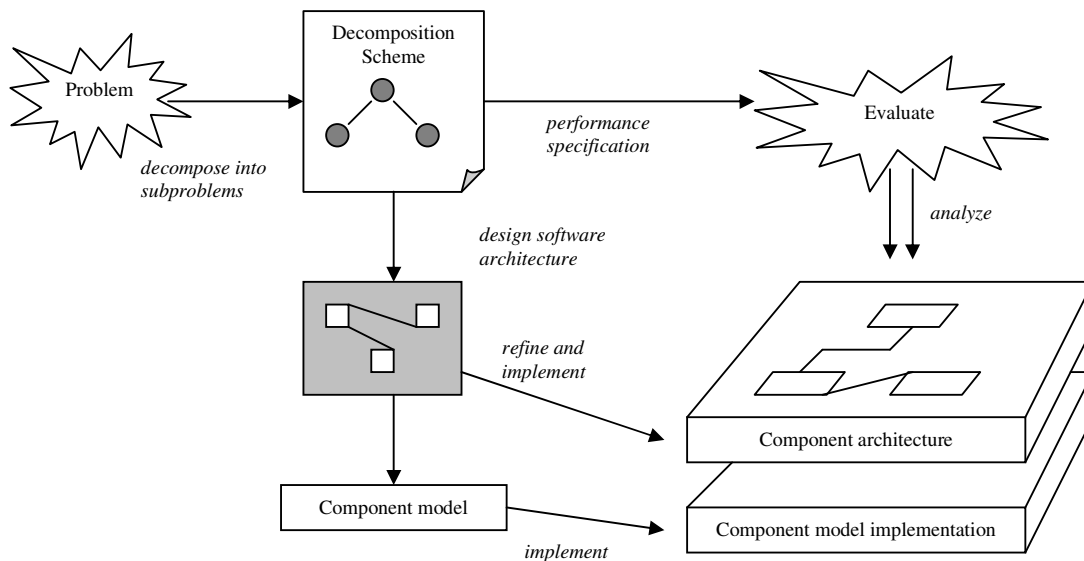


Figure 3: Paths followed in component-based software development

Software development is always initiated by recognizing a problem or desire and acting on it. Once the specific requirements and a high-level model of the software system that is going to solve the problem have been designed, the complex problem is recursively decomposed into smaller problems until the sub problems are elementary enough. The decomposition process results in a decomposition scheme. The software system is constructed by integrating the solutions of each of the sub problems bottom upwards until the entire software system has been integrated. The decomposition

scheme follows a tree-like hierarchy. Each of the leaves of the decomposition scheme can be assigned to different development teams, thereby speed up the development of the software system. In order for the integration to run smoothly the boundaries and dependencies between the sub problems must be clearly defined.

The software architecture is a derivative of the decomposition scheme. In general it can be said that each of the leaves in the decomposition scheme result can be mapped to a software component of the software architecture. The hierarchies in the decomposition scheme express the relationships and links between the different components. The software architecture comprises of the set of interacting components that is designed to ensure that the software system or sub system that are constructed from those components will meet certain constraints. The design of the software architecture is usually an iterative process in which the software architecture or parts of it are possibly implemented or simulated and evaluated. If the evaluation reveals that the constraints are not met the software architecture is redesigned.

The component model defines the specific interaction standards between the components and the composition standards for binding components together. The component model implementation is the set of executable software elements that are required to support the execution of the component conforming to the model. Examples of component models are for instance the Component Object Model (COM) [COM] by Microsoft and Javabeans [JavaBeans] by Sun.

2.2 Product Concept Assessment

2.2.1 About the assessment of concepts

Philips operates in a market with many competitors that are capable of producing qualitative consumer products. For Philips it is therefore important to differentiate their CE devices by incorporating nice advanced features to make them more attractive to consumers. Product innovation and development is therefore a key element. However it is difficult to say which features are useful and feasible. Furthermore, product development is a long and costly road that needs to be much faster to compete in the market.

2.2.2 Features and technologies

Research and development departments generate and develop new technologies for product innovation continuously. These technologies are usually designed specifically to solve or optimize a dedicated problem. In isolation these technologies are therefore not attractive, but in combination with other technologies they can be an answer to a useful problem; they become features. An example of a feature is to combine a technology that is capable of filtering out clear still images from a video stream with a face recognition technology to create photo albums out of personal holiday recordings. Product concept assessment deals with discovering and recognizing these useful features out of a big pool of available technologies. For this to happen a platform is required in which these technologies can be easily combined together and evaluated for there feasibility and attainability.

2.2.3 Dynamic Composition

The strict component-based development process as described in [2.1.2 Component-based software development] does not lend itself for the problem of product concept assessment. The development process follows a strict top-down approach in which the problem is decomposed into smaller problems and a software architecture is designed to meet these problem. Components are therefore bounded to each other in a very early stage and it is assumed that these bindings are more or less static, i.e. they will not change during the further development. Changes in the bindings between components in a later development stage therefore are not done straightforwardly, but require a huge effort.

Product concept assessment requires a flexible way for combining components. In the early stages the usefulness and feasibility of a certain combination of components is unsure and changes in the bindings of the components may prove to be necessary. Furthermore, although there is a general specification of the requirements to which the final software system has to conform and a rough decomposition sketch of the overall system is present, detailed specifications of the decomposition schemes and the software component architecture are not at hand. Seen from the viewpoint of product concept assessment it is more desirable to adopt a more or less bottom-up like approach of development in which components can be combined and “clicked” together in multiple ways. Due to the lack and uncertainty of detailed specifications of the software component architecture, the bindings between the components are determined through a kind of trial-and-error development approach in which the product concepts are prototyped and simulated on a real-time system.

2.2.4 Requirements for compositional approach

The development process as described in the previous section imposes certain requirements on the underlying component architecture and component model. Component architecture is defined as a set of platform decisions; a set of component models; and an interoperation design for the component models [Szyperski, 1997]. It defines the way in which components and component models can be installed onto the platform and specifies the set of interfaces and rules of interaction that govern the communication between components. Another important requirement for a compositional approach for designing software systems is to have a flexible component model. A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution, and deployment [Heineman & Councill, 2001]. For the assessment of product concepts and the prototyping and simulation of these concepts the following requirements are necessary:

- **Dynamic composition:** Many product concepts can be derived from the components that have been installed on the platform. In order to explore this pool efficiently components must be bounded in a late stage and the bindings between the components must be easily changed. The component architecture and the component model must support this dynamic composition.
- **Flexible platform:** Components can easily be added and removed from the platform. Furthermore as components themselves are in a development stage

the platform also needs to support the substitution of components without the need for reconfiguration of the platform.

Szyperski [Szyperski, 1997] observes that a great “gap” between the virtual platform (platform abstractions) and its underlying platform generally has a tremendous impact on the expected performance. Understanding this distance is very important and thus it is essential that the concepts of the component system architecture can be mapped efficiently to the underlying component model and physical platform.

2.3 STAGE Prototyping

Within the research and development at Philips Research many new great technologies are developed continuously. It has been recognized that a platform is required that enables an easy way for developing software systems out of these technologies and prototyping of these systems. For this purpose the STAGE platform has been devised.

2.3.1 Focus of STAGE

The STAGE platform [Lange & Kang, 2004] provides an environment in which architecture and product concepts can be rapidly prototyped and simulated in an architecture-true way. The STAGE platform relies on PC's and PC networks, because PC's provide the performance and flexibility that are required to enable rapid prototyping. Furthermore the generality of the PC platform allows the usage of state-of-the-art PCI plug-in media processing boards and advanced libraries that are widely available on the PC platform.

The focus of STAGE is in the component interaction architecture, i.e. in the subdivision of the total system in hardware and software components and their interactions. By making a well-defined separation between platform independent and platform dependent components architecture-true prototyping becomes possible.

2.3.2 System Architecture

A common used method for representing software architectures is a layered model in which each layer relies on the services offered by the layer beneath it and provides services to the layer above. Figure 4 shows an example of software layers that are relevant for the STAGE platform in the context of the leading example.

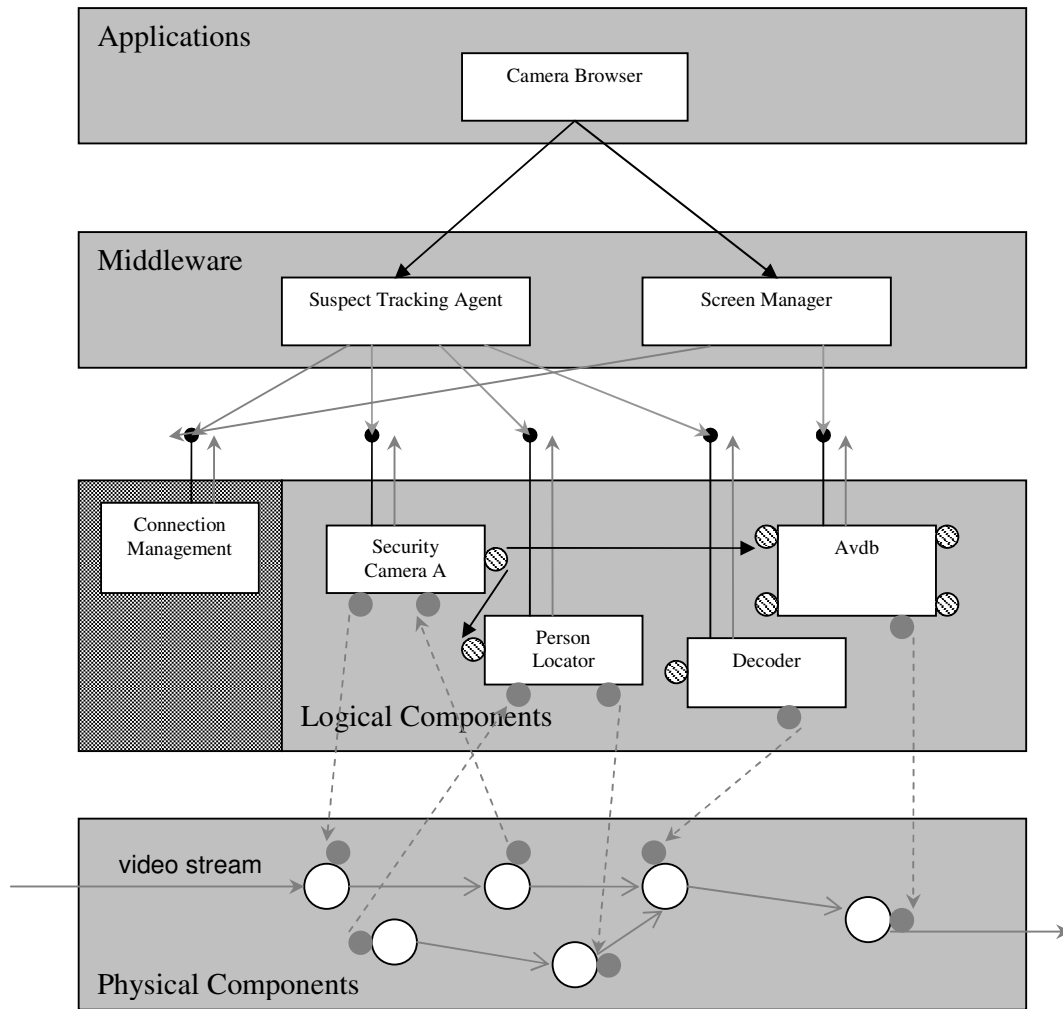


Figure 4: Software architecture as stack of layers

Each of the layers and the entities in these layers as depicted in Figure 4 is described in more detail in the following sections. The description goes from the bottom layer to the top layer.

2.3.2.1 Physical Components

The physical component layer consists of the software and hardware entities, which will be called tasks that perform the actual streaming operations. Tasks are considered to be units of specification and generally consist of low-level signal process functions, like variable length decoding, DCT calculations. The media processing system is modeled in this layer as a process network, which makes communication and parallelism explicit. For more details see paragraph 3.7. A sequence diagram that specifies the interaction between tasks has been included in Appendix 8.6.

2.3.2.2 Logical Components

An important concept within STAGE is logical component. A logical component is a high level abstraction of a media processing functionality. The concrete pieces of implementation of a logical component are the tasks in the physical components layer. An example of a logical component and the tasks it relies on is for instance the

MPEG2 decoder component as depicted in Figure 6. Three logical components are shown in Figure 4: a SecurityCamera components stream out the images captured by the security camera, a person locator determines the location of a person in a stream, a decoder that shows an incoming stream on a separate screen and an Avdb that records streams and plays recorded.

Logical components do not process data streams, but merely implement the control part of the streaming processing. Middleware applications access the functionality provided by the logical component through well-defined interfaces. Each interface captures a separate aspect of the logical component. This separation improves maintainability of the logical component. The SecurityCamera component can for instance provide an interface through which the angles of the camera can be adjusted while another interface controls the zooming. Associated with each interface is a notification interface through which the logical component can notify middleware applications of certain events. In Figure 4 it can be seen that data streams only reside in the physical components layer and do not flow through the above layers. Logical streaming connections are created by connecting pairs of pins and they can be efficiently mapped to the underlying physical streaming connection. The logical component and the physical components they encapsulate are explained in more detail in [3.2 Logical Component].

2.3.2.3 Connection Management

The STAGE platform is about assessing product concepts. One must to enable this assessment is to have a way to easily recombine and integrate the logical components to form new behavior and functionality. A specialized component called the Connection Manager is responsible for instantiating logical components and setting up streaming connection between the pins of logical components. The connection manager is explained in more detail in the following chapter.

2.3.2.4 Middleware

The middleware applications in the middleware layer mainly provide a high-level abstraction of the functionality provided by the logical components. A middleware application can be mapped to one or more logical components in the logical components layer. An example of a middleware application is for instance the Suspect Tracker Agent which relies on the SecurityCamera component, the PersonLocator component and the decoder component. This middleware application automatically follows a suspicious person by adjusting the viewing angle of the security camera to the coordinates determined by the PersonLocator component. During these adjustments the video stream from the SecurityCamera component can be seen in real-time from the screen

2.3.2.5 Applications

A typical application in the application layer can for instance be a Camera Zapper for selecting the video stream of a security camera to be shown on the screen. For applications to function they rely on the functionality of the middleware applications.

3 ADVANCED CONNECTION MANAGEMENT

3.1 Introduction

The advanced connection management (ACM) defines an architecture to manage the complexity of developing, building, prototyping media processing systems in a distributed manner. Media processing systems are constructed within the ACM by dynamically composing different software components together. Software components in the ACM are deployable software applications that run distributed on machines in the network. The high-level abstraction provided by the ACM enables an easy and fast way for developing media processing systems through the dynamic modeling and building of software component compositions. At design time the interconnections between the components of which a media processing systems are not (fully) defined. Instead a generic description of the software component composition is provided at run-time from which the software component composition can be generated and built. The ACM allows for components to be easily “plugged into” and “unplugged from” without the need for reconfiguration.

An important part of the ACM is the specification of an underlying component model. The component model specifies and governs the rules of interaction between the components and provides mechanisms for the composition of components. The component model is designed in such a way that the high-level abstractions of the ACM can be efficiently mapped to the underlying entities in the component model and the physical platform.

3.1.1 Goal

The fundamental goal of the ACM is to enable a way to rapidly develop and build media processing systems and to analyze the potential and feasibility of these systems. The approach taken by the ACM is to build media processing systems from software components, which will be called logical components, that are deployed and running distributed over machines in the network. These logical components are the fundamental building blocks for media processing system integration. The high-level abstraction provided by the ACM hides the distribution of the logical components and the physical network from the media processing system.

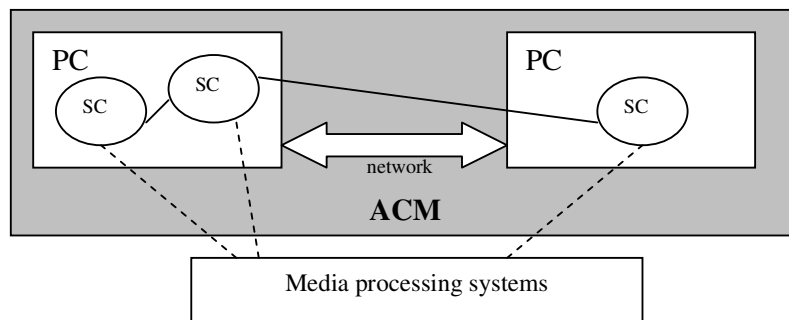


Figure 5: Constructing applications from deployable software parts

The ACM allows media processing systems to change their behavior and functionality during execution by dynamically changing the logical streaming connections between

logical components. The media processing system is in principal nothing more than an over coupling application that supervises the logical components and that uses the functionality provided by the logical components to accomplish its job. The ACM provides the functionality to dynamically setup different compositions of logical components. This chapter explains the design and the realization of the ACM.

3.2 Logical Component

The logical component in the STAGE platform is the basic building block of a distributed media processing system. From the perspective of the middleware applications logical components are viewed as “black-boxes”. Communication of middleware applications with the logical components proceeds through standardized interfaces. The logical component is an abstract notion that exposes useful functionality to middleware applications. Tasks (see 3.2.2) form the concrete pieces of software or hardware that processes and transform streams, while the logical component is the high-level abstraction characterized by the abstract set of media processing functionality. An example of a logical component is for instance a MPEG2 decoder. Figure 6 shows the streaming network of an MPEG2 decoder. Decoding a MPEG2 streams consists of multiple steps such as variable length decoding, run-length decoding, motion compensation, etc.

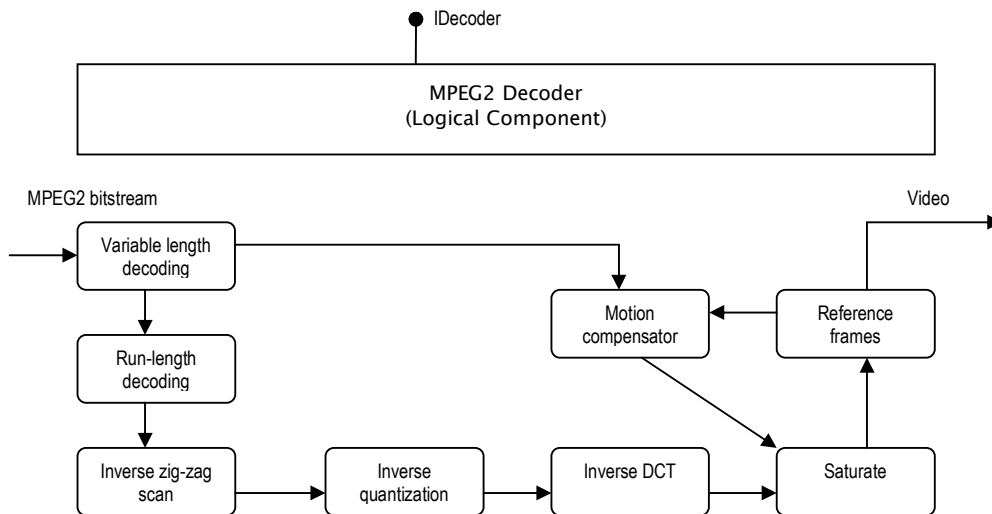


Figure 6: MPEG2 video decoding

From Figure 6 it is apparent that MPEG2 video decoding is a complex process. This complexity is hidden from a client application through the MPEG2 decoder logical component abstraction. Client application communicate with the MPEG2 decoder through the `IDecoder` control interface implemented by the logical component, which contains simple methods as `startDecoding()` and `stopDecoding()`. Each logical component has a certain type. An instance of the MPEG2 decoder logical component is of type MPEG2 decoder. Due to the abstraction of the logical component each instance of a logical component is interchangeable with another instance of the same type regardless of its implementation. Although the implementation may have changed the logical component abstraction and its interfaces remain the same. Logical components are independent of programming languages and platforms, while their implementations are not.

3.2.1 Communication with outside world

In order to easily combine logical components together and to realize modularity and replaceability the communication of logical components with other logical components and the outside world is restricted to well-defined standard mechanisms. Beside the control interfaces and the notification interfaces associated with each control interface (see 2.3.2.1 Physical Components) for communicating with middleware applications, each logical component has a predefined set of pins. Pins are the linking points of connections and are the only means to communicate streaming data between logical components.

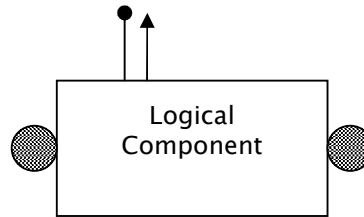


Figure 7: Visualization of logical component

Pins can be separated into two kinds: input pins and output pins. Through their input pins logical component receive streaming data, while data streams flow out of the output pins of a logical component. Each output pin of a logical component can be logically connected to the input pin of a logical component. Each pin of a logical component is associated with one or more data types. A logical streaming connection between two pins can only be created when the pins have compatible types.

3.2.2 Task

A task is defined as the basic processing entity that implements a certain specific signal processing function. An example of a task is for instance the variable-length coding task as shown in Figure 6. Tasks have predefined input and output ports and ports are the only means of a task to communicate with its environment. By restricting communication on the ports the communication mechanisms of the task are kept symmetric. Not only does this increase the modularity of the task, but it also aids in porting the task to an embedded system. Tasks consume data from their input ports, process the data and produces data on it output ports. Each port of a task is associated with a certain type and tasks expect streams of this type. Two kinds of streams are handled by a task: control streams and data streams. Control streams are sent by the logical components to the task and affect the behavior of the task or notify the logical component of a certain event. A typical control action is for instance stopping the processing of a task. Typically tasks process their streams in an infinite loop which is executed in a separated thread. Generally this processing loop looks as follows:

```

while (true)
{
    Frame f;
    streamportA→read(f);
    PROCESS THE FRAME
    streamportB→write(f);

    ControlCommand cmd;
    controlportA→read(cmd);
    if (cmd == STOP)
    {
        controlportB→write(STOPPED);
        return;
    }
}

```

The processing of control commands can only occur on certain reconfiguration points in the processing loop. This is specific to the signal processing function. For stopping a MPEG2 decoder task it would make sense to only stop at I-frames. Reconfiguration points must not be too fine grained, as the overhead introduced by the checking would be too great. However a coarser grained approach degrades the responsiveness of the task to control commands.

The signal processing function implemented by a task can be implemented in software or in hardware. If the task is implemented in hardware, the task encapsulates the hardware task providing well-defined ports for the communication with other tasks and its environment. This encapsulation has been illustrated in Figure 8. The stream consumed by the task through its ports is processed by a hardware module and subsequently produced to the tasks outputs.

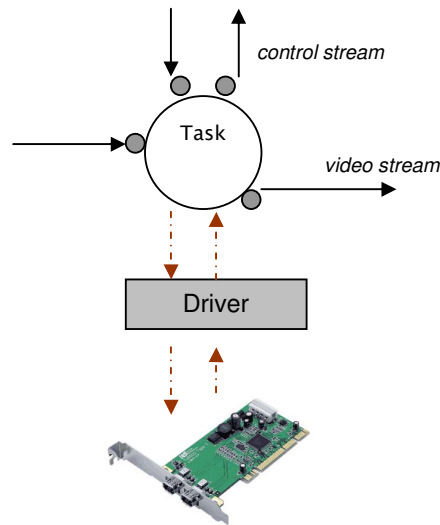


Figure 8: Task encapsulation of hardware

3.3 Use Case

A use case in the ACM is defined as a possible configuration of a media processing system and represents the applications of this configuration. The definition of use case differs from the common use case definition in the Unified Modeling Language (UML). A possible use case of the security monitoring system is for instance the view of a recorded stream from the audio video database. A intuitive way for representing a

use case is by a stream graph with directed edges that on a high level indicates the streams that flow through the logical components involved in the use case. Figure 9 shows the stream graph for the mentioned use case.

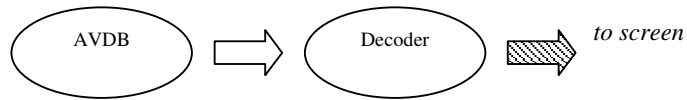


Figure 9: Stream graph

The media processing system represented by the use case typically requires the functionality of certain logical component and relies on a certain composition of the logical components. The software component composition defines the interconnection between the logical components. At the middleware applications level the specifics about the composition are not known and setup of the software component composition proceeds implicitly.

Beside the setup of the software component composition a use case usually also entails configuring the behavior of the involved logical components to fit the use case. These configuration actions are typically application specific. An example of such a configuration action is for instance which recorded stream should be played from the AVDB in the above described use case. The ACM provides functionality for setting up the required software component composition, but application specific configuration actions are necessary to instruct the logical components.

3.3.1 Use case description

A use case description contains a generic description of a media processing system. Generic descriptions are extremely useful for modeling dynamic compositions of complex media processing systems as they specify in a high-level how the software component compositions can be generated “on-the-fly”. It is the responsibility of the ACM to build the contained software component composition description by activating the appropriate logical components and create the appropriate connections between them. With generic description files it is possible to specify complex hierarchies of software components through for instance compound components. Compound components are components that represent a composition in the small and thus raise the level of abstraction. For reusability reasons it is essential that the description is kept as simple as possible. The intelligence that is required for setting up the use case should not be contained in the description itself, but should be inherent in the ACM.

The description of a use case is noted in the eXtended Markup Language (XML) [XML, 2000]. XML is a markup language that presents information in a structured way. The meaning of the markup language is described in the Document Type Definition. By denoting the description file in XML the possibility arises for external applications to generate or process the description file. A graphical environment in particular would be very useful for modeling the software component architecture at a very intuitive and high-level. An example of a XML description file of a use case can be found in appendix 8.1.

3.4 Connection Manager

The connection manager is an executable software component. Logical components can be easily plugged into and unplugged from the ACM. The connection manager implements high-level mechanisms and rules of interaction to dynamically build and reconfigure software component compositions. The connection manager must meet the following requirements:

- Components can easily be plugged into the ACM and also be able to be unplugged from it without the need for reconfiguring the connection manager.
- The connection manager holds detailed knowledge about the capabilities of the ACM. The capabilities of the ACM are dynamically formed by the abstract set of functionality provided by the logical components that have been plugged in. The capabilities of the ACM change when components plug into or unplug from the ACM.
- Set up the software component composition that is required for a certain use case by creating the appropriate connections between the logical components. The software component composition is built from a generic description of it.
- Validate if a certain software component composition can be setup. Among many things this comprises the check if the required logical components are available. Furthermore it is also required to determine if connections between logical components can be setup.
- Dynamically reconfigure a software component composition into another without stopping the flow of streams. The issues involved in dynamic reconfiguration are addressed in detail in chapter 4.
- The connection manager functions as a kind of a mediator thereby decoupling the middleware applications from the logical components. The connection manager holds an overview of the logical components that are plugged into the ACM and middleware applications request the connection manager for interfaces to particular logical components. The connection manager hereby acts as a mediator between the two.

The connection manager is regarded as a service and its functionality is exposed to middleware applications through an interface. Each middleware application holds an interface to the connection manager. This interface is described in more detail in 3.4.2 Logical Connection Manager.

3.4.1 Separation into logical and physical

In the layered model as depicted in Figure 4 a clear distinction between the logical component layer and the physical component layer is visible. This separation allows for a high-level abstraction. If the connection manager were put into context of this layered model it would result in figure below.

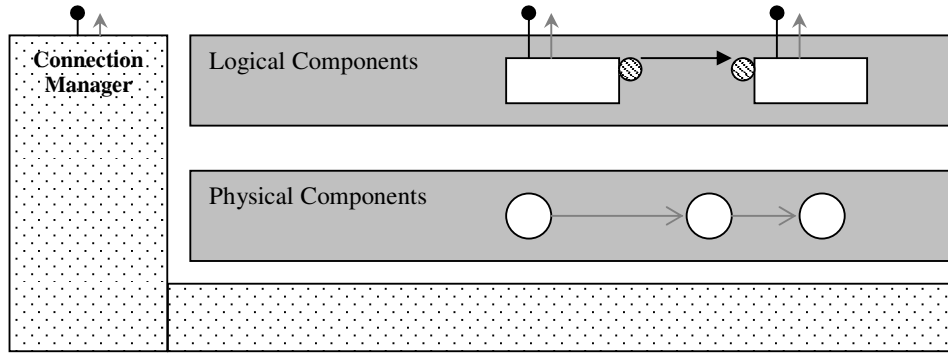


Figure 10: Position of connection manager in layered model

From Figure 10 it becomes apparent that the connection manager spans both the logical component layer as the physical component layer. The connection manager in principle runs as a service beside both layers and both layers are built on top of this architecture. The spanning of the connection manager across both layers however has some drawbacks. The connection manager operates both on a high level with the logical components and the logical streaming connections between them, but also requires to have a thorough understanding of the underlying streaming infrastructure. Although this approach straightforwardly allows for an efficient mapping of logical streaming connections to physical streaming connections, it sacrifices flexibility. Changes in the connection manager at some level can have consequence that run through the entire advanced connection manager. For instance a change in the streaming technology used for the transport of streams can affect the way logical streaming connections are mapped or setup.

A greater abstraction and separation of concern can be reached if the connection manager is divided between the logical component layer and the physical component layer as has been illustrated in Figure 11.

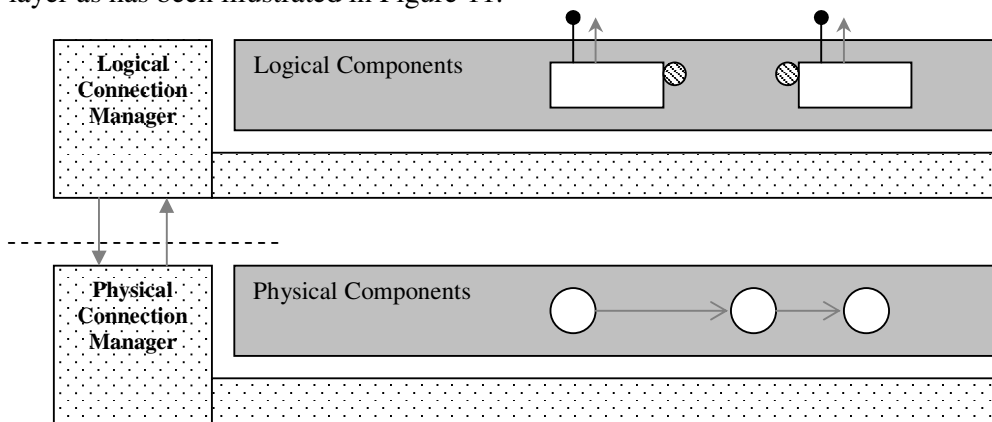


Figure 11: Division into logical and PCM

The connection manager is separated in a logical and a PCM that interact with each other through standardized interfaces. In this way changes made in either the LCM or the PCM has only a local effect as long as the interface remains compatible. One has to keep in mind however that this separation does not take into account of the efficient

mapping of streaming connections. Both the LCM as the PCM are explained in more detail in the following sections.

3.4.2 Logical Connection Manager (LCM)

The LCM is an executable software program that implements the functionality to setup compositions of logical components, i.e. the connections between them that are required for the construction of media processing systems. The LCM holds an overview of the logical components that have been plugged into the ACM. In practice it is the case that the LCM, the logical components and the media processing systems do not run in the same process or machine. A remote proxy pattern [Gamma et al, 1995] is used to hide the physical interconnection platform. The ACM can therefore be more easily deployed on different platforms. Each interface of a logical component is associated with a proxy / stub pair. The proxy acts as a surrogate for the actual logical component and implements the exact same interface as provided by the logical component. Calls that are made to the proxy are forwarded to the associated stub that makes the actual call on the interface of the logical component.

Middleware applications require the logical components for operation. Therefore a lot of dependencies exist between them. To decouple the middleware applications from the logical components the LCM acts a mediator between the two by restraining both entities from explicitly referring to each other. Interfaces to logical components are supplied to middleware applications on request. These interfaces are implemented by proxies, which are created through a factory pattern [Gamma et al, 1995]. Proxies are created by the LCM through the proxy factories that are registered to the LCM.

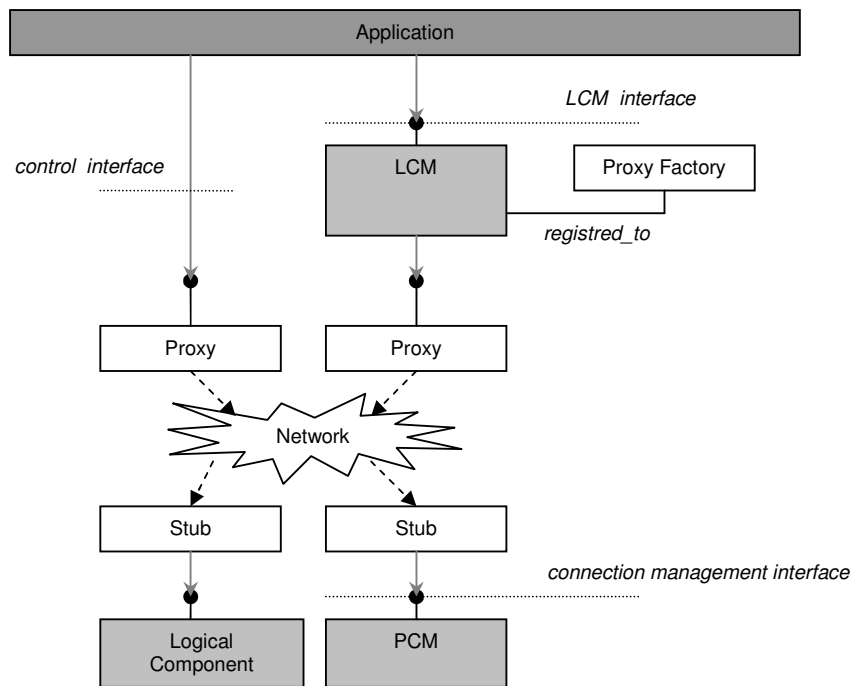


Figure 12: Software architecture LCM

The LCM implements the high-level functionality for setting up the compositions of logical components. This functionality is exposed by the LCM to the middleware

applications through the `ILogicalConnectionManagement (ILCM)` interface. It is important that the exposed interface does not contain any references to the underlying connection platform. Details and intelligence must be contained within the LCM. The `ILCM` interface contains methods to setup software component compositions from use case descriptions and methods to obtain interfaces of logical components.

A software component composition is setup by the LCM by creating the appropriate logical streaming connections between the pins of logical components. Connections are created by the LCM through the communication with the physical connection managers (PCM) that are local to the logical component. The communication between the LCM and the PCM proceeds through a standardized connection management interface (CMI). This interface abstracts the underlying streaming technology from the LCM. The PCM and the CMI are explained in more detail in the next section.

3.4.3 Physical Connection Manager (PCM)

Logical streaming connections indicate the logical flow of data between the pins of the logical components. The LCM creates logical streaming connections. The actual streaming connection between the logical components however depends on the underlying streaming technology used and can traverse along a completely different route than the logical streaming connection suggests. Figure 13 for instance illustrates an example where the streaming data needs to pass different network layers and over the physical network before it reaches the other logical component. From the client application's perspective however the connection appears to be direct.

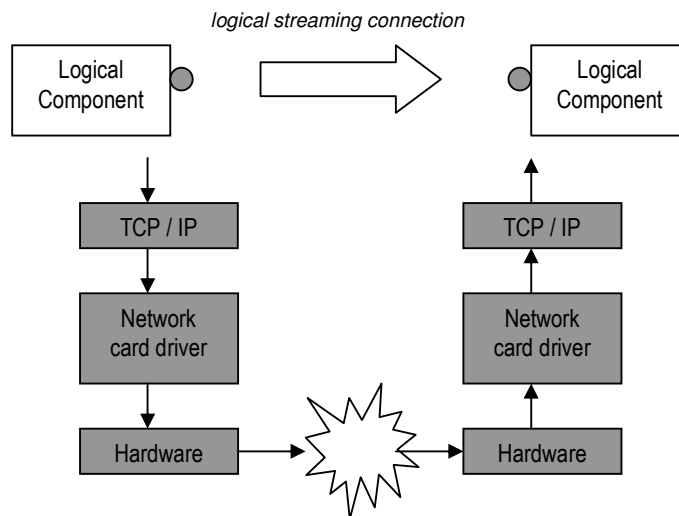


Figure 13: Streaming implementation

To shield the underlying streaming technology from the LCM a PCM that is local to the logical component takes care of this. This allows for a better separation of concerns. The association of the PCM with the logical component has been implemented through inheritance: the `LogicalComponent` class is derived from the `PhysicalConnectionManager` class. The PCM is exposed to the LCM as an interface of the logical component as has been illustrated in Figure 14.

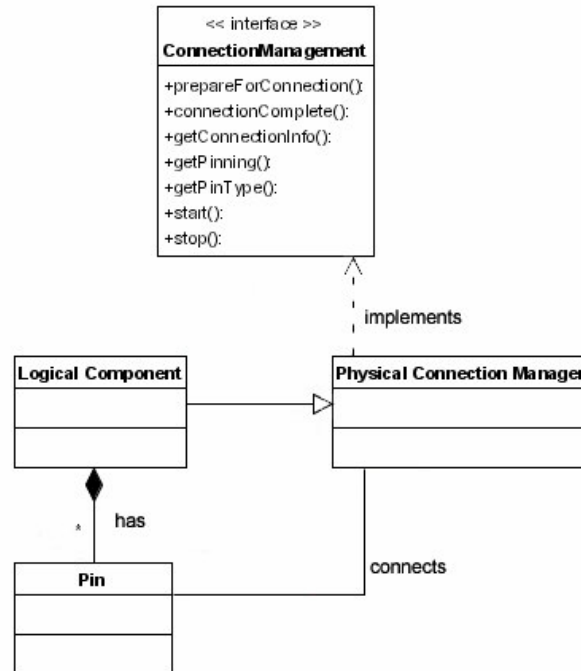


Figure 14: Class diagram CM

3.4.3.1 Connection Management Interface (CMI)

The Connection Manager interface allows the LCM to use and manage connections between logical components. It abstracts from the physical interconnection technology. Figure 14 shows the most common methods in this interface and it provides the LCM with a mechanism to do the following.

1. Retrieve detailed information about the logical component, such as the pinning.
2. Perform capability matching between two pins. This involves for instance:
 - a) Content-format matching: Matching of the type between the output pin and the input pin. Connections can only be setup between pins that support the same type.
 - b) Streaming protocol matching. Checking if the PCMs of both the logical components support the same streaming technology.
3. Create and teardown connections between logical components.
4. Reconfigure the connections that have been created on the pins of logical components.

The Connection Management interface conforms to the Connection Management service as set by the UPnP AV framework [Chan, 2002]. Conformance to this standard simplifies the integration of the logical components into other UPnP architectures and also allows the use of tools for UPnP technology within the STAGE platform, such as the Intel's Device Spy [IntelDeviceSpy].

3.5 Dynamic Plug-in Architecture

The architecture defined by the ACM provides support for the dynamic plugging of logical components. To this end the LCM maintains a repository that contains all the logical components that have been plugged into the ACM. This repository changes when logical components are plugged into and unplugged from the ACM. Each logical component in this repository is identified with a unique identifier.

Plug and unplug operations are decided upon and initiated by the logical components. The logical connection has no control over the moment of plugging or unplugging of logical components, but is only notified of the event. Two methods have been defined for the plugging and unplugging of logical components:

- `logicalComponentPlugged(string id)`
- `logicalComponentUnplugged(string id)`

The lack of control of the LCM of the plugging and unplugging of the logical components introduces certain problems. Situations can occur where a logical component that is part of a running use case decides to unplug itself from the ACM. As a consequence the use case is broken and needs to be fixed by the LCM by bringing the media processing system into a consistent state. An option would be to restore the use case by replacing the unplugged logical component with a compatible logical component. It is assumed an external entity (possibly the user) is responsible for the correct moments of plugging and unplugging of logical components, although the LCM needs to be able to cope with these situations. These situations are considered to be out of scope of this thesis.

3.5.1 Self-Describing Logical Components

One of the requirements for allowing dynamic and flexible plug-in architecture for logical components is that logical components are self-describing. The LCM should have no pre-defined information about the logical component, but all required information of the logical component must be obtainable from the logical component itself. Self-describing logical components do not only allow for a looser coupling, but also the description always reflects the current state of the logical component. Information of a logical component can be retrieved by the LCM through the CMI (see 3.4.3.1).

3.6 Dynamic Connection Management

3.6.1 Dynamic Interconnection

The ability to change connections at run-time is essential for the easy assessment of product concepts and distributed media processing systems that implement different use cases. The LCM is responsible for the dynamic reconfiguration of the interconnections between the logical components. A well-known property within database programming is ACID. By conforming to the ACID properties a consistent state of the media processing system can be ensured.

The following can be said when viewing ACID in the perspective of dynamic reconfiguration.

- **Atomicity:** each reconfiguration should be carried out as a single operation. An atomic reconfiguration ensures that either all changes of the reconfiguration occur or none at all. Basic reconfiguration operations however should not be restricted to this property. Reconfiguration operations in the ACM proceed in an asynchronous manner to benefit from the parallel execution of these operations. A great of this benefit gets lost when atomicity is imposed on a basic reconfiguration operation. Therefore only the entire reconfiguration must adhere to atomicity.
- **Consistency:** every reconfiguration should take the media processing system from one consistent state to another. To some extent a consistent state is dependent on the media processing system. The security monitoring system can for instance specify that each security camera be at all times connected to at least one decoder.
- **Isolation:** Reconfiguration operations are isolated from one another. Race conditions must ensure that no situations can occur in which a cross dependence of resources by different entities is required. These situations can lead to deadlock. Before the LCM starts a reconfiguration process it first claims the resources that are needed to do the reconfiguration and if not successful releases them all.
- **Durability:** Not really applicable to the situation.

The LCM implements the basic reconfiguration operations that are necessary for performing dynamic reconfiguration.

3.6.1.1 Connection

A connection is a relation between the pins of logical components and is defined as a pair of pins (pin_1, pin_2) where $type(pin_1) = type(pin_2)$ and pin_1 and pin_2 are an output pin and an input pin respectively. For the LCM a connection is not a tangible entity, but is an abstract definition that merely states the relationship between two logical components. The LCM provides a basic reconfiguration operation `createConnection()` for creating a connection between two pins of logical components. The creation of a connection consists of preparing the connection at both pins of the connection. A `prepareForConnectionOnPin(pin1, pin2)` is defined in the connection management interface that connects pin_1 of the logical component to pin_2 . A connection (pin_1, pin_2) means that pin_1 is connected to pin_2 and pin_2 is connected to pin_1 . At any point it must be valid that if pin_1 is connected to pin_2 than also pin_2 is connected pin_1 . The `createConnection()` operation is an atomic reconfiguration operation, i.e. either a connection is setup or no connection is setup. A `teardownConnection()` operation is defined by the LCM to remove a connection between two pins. Once the connection has been actually established, a notification is sent for this event to the LCM by both the PCMs that were responsible for setting up the actual connection.

3.6.2 Mapping from logical to physical

The LCM allows for an easy and flexible way of developing and constructing media processing systems by providing a high-level abstraction. This high-level abstraction enables a dynamic compositional approach for building software component compositions. However, a commonly observed phenomenon is that obtaining a high-level abstraction often goes to the expense of performance. Viewed from the point of reusability and separations of concerns having a clear separation between logical streaming and the actual implementation of the streaming is important. By associating a local PCM with each logical component, the LCM is completely abstracted from the underlying streaming technology. The PCM handles the specifics of the setting up the actual streaming connection. This section describes the mapping of the logical streaming connection to the actual streaming connection.

3.6.2.1 Creation of a logical streaming connection

Transformations on streams are conceptually performed by the logical components and logical streaming connections indicate the flow of streams throughout the logical components. Connections between the pins of logical components are created by the LCM through the interaction with the PCMs. The PCM is responsible for the setup of the actual connection on a local basis. Figure 15 depicts the mapping of the logical layer to the physical layer and it is explained how the actual streaming connection is setup by the PCM.

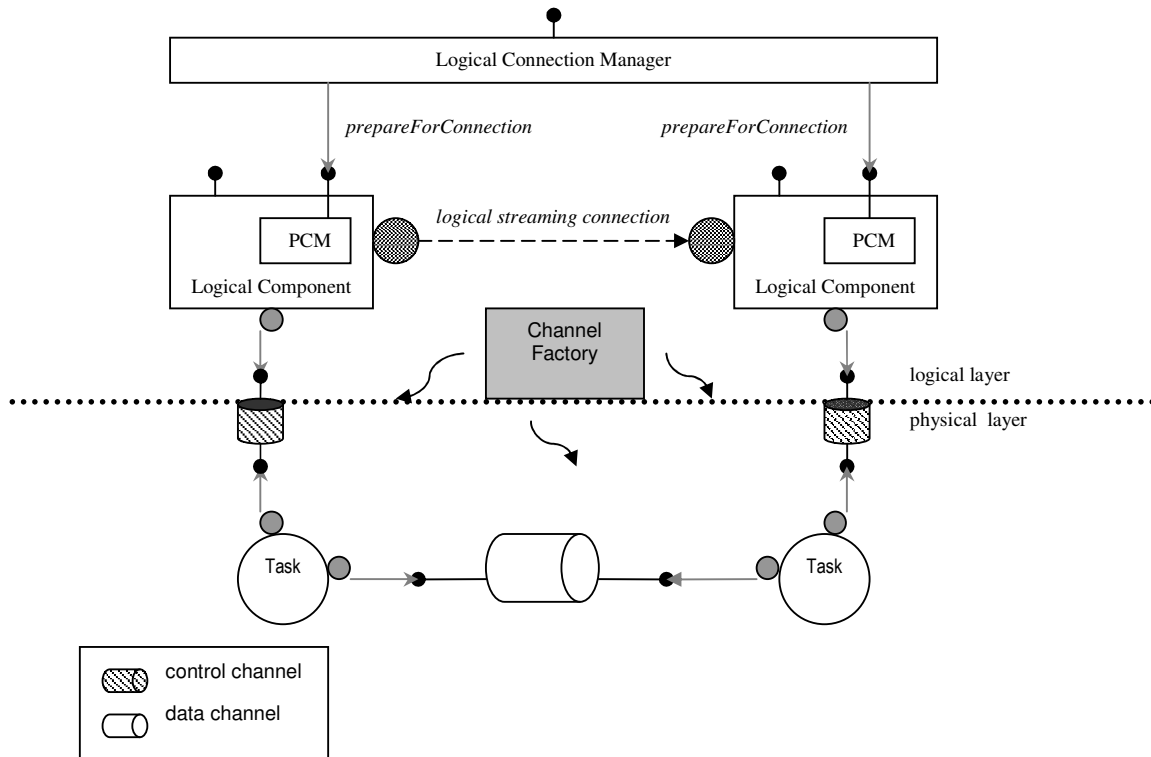


Figure 15: Logical streaming connection and actual streaming connection

Tasks are the actual entities that perform the transformation on streams. Channels are used to model the streaming communications with and between the tasks and they provide a simple read / write interface. Channels abstract from the physical streaming

platform (see 3.7.2). The PCM is responsible for setting up the underlying streaming connection and it does so by creating a channel. Channels are created by the PCM through a factory pattern. By centralizing the knowledge of creating channels within the channel factory, a decoupling between the PCM and the streaming technology is reached. This decoupling provides an easy way for the use of different channel types. Internally channels can for instance use different transportation protocols (TCP/IP, RTP) or have different communication styles. A streaming connection is created by coupling the interfaces exposed by the channel to the ports of the tasks. The setup of a logical streaming connection is explained in more detail in the sequence diagram in appendix 8.4.

3.7 Model of computation

A model of computation is a virtual machine with states and computational steps, which provides a foundation for reasoning and analyzing algorithms and systems thereby ignoring many implementation details. The model of computation provides a model of concurrency and describes the interactions between the computational elements.

3.7.1 Kahn Process Network

A frequently used model of computation used for signal process functions is the Kahn process network [Kahn, 1974]. The Kahn process network represents the different steps of media processing through tasks communicating with each other through unbounded First-In-First-Out (FIFO) channels. It makes concurrency and data buffering explicit. The explicit concurrency of the Kahn process network makes it to fit well with the distributed architecture of the STAGE platform formed by the PC's and the PC networks. The only means for tasks to communicate with each other is through channels. Two operations are defined on a channel: read and write. A read action on an empty channel is considered to block until a write action is performed on that channel. Write actions on channels never block as channels have unbounded capacity.

3.7.2 Streaming Model

In realistic applications channel capacity is limited. This means that when channels are full, data will be overwritten or spilled. Therefore a slight variation is made to the model: writes on full channels are blocking. By bounding the capacity of the channels artificial deadlocks can be introduced [Parks, 1995]. Figure 16 gives an overview of the underlying streaming model of the ACM.

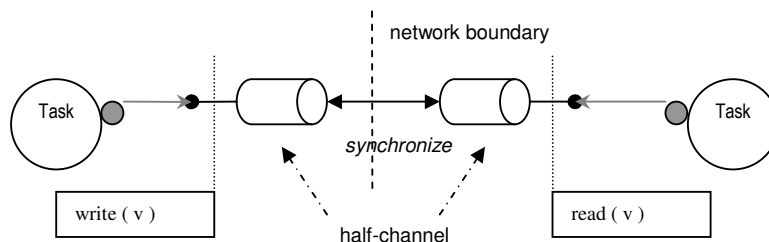


Figure 16: Underlying streaming model

In contrast to the Kahn process network tasks do not operate directly on the channels, but communicate only on their ports. Ports are the static communication points for the task to dynamic channel structures. During reconfiguration of the process network ports of tasks can be reconnected to different channels without the need for destroying the task. Access to the channel structure is modeled as an interface. The communication style of the channel consists of a simple read and writes methods. Channels are directed and therefore the interface is split into a write and read interface that are coupled to output and input ports respectively. The ports and the channels abstract the physical streaming technology from the tasks, making them easier to port.

3.7.2.1 Half-Channel

A commonly observed phenomenon in a distributed system is the added latency that is introduced through remote procedure calls. Resources can be required by tasks that are physically in a different location. Furthermore heavy network traffic can influence the behavior of the media processing system. Consider for instance the security monitoring system in which the different streams captured by security camera's come together in a central control center. The security monitoring system is a distributed system as the security camera's and the control center are in different geographical places. It is evidently essential that the video streams captured by the security cameras must be shown within a reasonable amount of time on the screens in the control center without any side effects, such as freezes due to lack of streams. Blocks of tasks that occur due to the distributed nature of the resources must be minimized. For this reason the half-channel concept has been introduced.

In general there are two straightforward possibilities for the location of the channel in comparison with the producer and consumer it connects. It can be located local to either one of the tasks or it can be located remote from both tasks. The half-channel concept splits the channel into two parts that are located local to each of the tasks. Each channel is thus represented by two half-channels. A synchronization procedure tries to keep the two half-channels consistent. The synchronization procedure has been implemented by two threads. The purpose of the half-channel is not to avoid blocks of tasks that are the result of the process network, but to avoid blocks of tasks due to the distributed nature of the resources. Furthermore as the network boundary falls exactly between the two half-channels it also aids in the reconfiguration of the process network as is explained in 4.4. A formalization of the half-channel can be found in appendix 8.6.1.

3.8 Universal Plug-and-Play (UPnP)

UPnP technology provides an architecture for peer-to-peer network connectivity and has been designed to bring easy-to-use, flexible, standards-based connectivity to ad-hoc or unmanaged networks [UPnP, 2003]. UPnP technology provides a distributed, open networking architecture that relies on web technologies such as IP, TCP, UDP, HTTP and XML. The UPnP architecture is designed to be to support “zero configuration” in which devices can join and leave the UPnP network dynamically without requiring any configuration whatsoever. The UPnP architecture is used within the ACM to provide a straightforward abstraction of the physical network. The properties of UPnP, like zero-configuration, open distributed networking architecture and abstraction from the physical locations of devices and control points can be used

in the ACM. This paragraph describes the UPnP architecture and explains how the UPnP architecture has been incorporated within the ACM.

3.8.1 UPnP Architecture Overview

The UPnP architecture defines two general concepts: (controlled) devices and control points. Devices can be seen as servers that provide some services to their clients, the control points. The device responds to requests from the control point and notifies control points of certain events. The working of devices and control points is explained through 5 steps of UPnP networking

1. Addressing
Each device must have a Dynamic Host Configuration Protocol (DHCP) client and obtain an IP address from the DHCP server. If no such server is available Auto-IP is used to acquire an IP address.
2. Description
When a device enters an UPnP network the device sends out notifications of its presence and advertisement for the services it has to offer by sending out device and service description expressed in XML. Control points are able to intercept these messages and thereby detect the devices and discover the services of a device. When the control point is lacking some device or service the control point can send out query messages for these devices or services onto the network.
3. Control
Control points can subscribe themselves to the services for which they have received advertisements. Once subscribed actions of the service can be invoked.
4. Eventing
Devices are able to inform control points of certain events by sending out notification messages.
5. Presentation
A device can have a presentation page, which can be retrieved from an URL and loaded into a browser. This enables a user to control and view the status of the device from an easy and independent point.

More detailed information about the UPnP device architecture can be found in [UPnP, 2003].

3.8.2 Simple UPnP Framework

UPnP technology is an open networking architecture and as such many different developers have implemented their own version of the UPnP stack. Examples of some of these developers are Siemens and Intel. Each of these developers provides their own Software Development Kits (SDK) using their own particular interfaces.

The STAGE platform is based on standard PC's, which can run either the Windows OS or the Linux OS. Due to the fact that no C++ SDK from a developer has been found that supports both the Windows and Linux, it was decided that a simple framework was needed to make the use of the UPnP stack transparent to the STAGE platform.

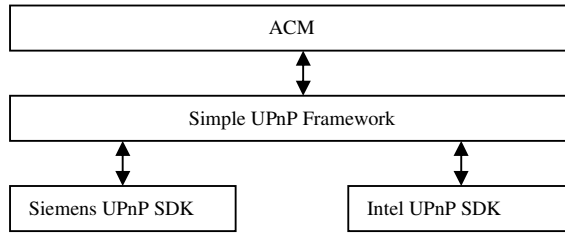


Figure 17: Simple UPnP framework

The Siemens C++ SDK [UPnP_Siemens] has been used for the Windows OS. Intel for instance provides a C++ implementation of the UPnP stack, but has not been used.

3.8.3 UPnP and ACM

This section describes how the UPnP architecture has been incorporated into the ACM. Figure 18 gives an overview of this.

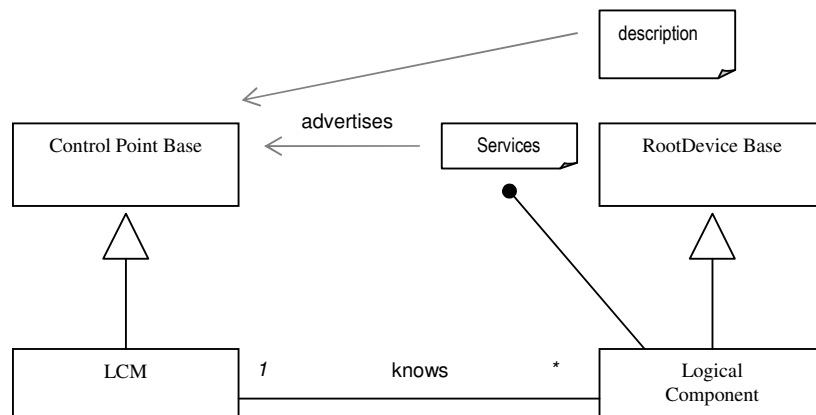


Figure 18: Control point encapsulation of LCM

3.8.3.1 LCM

A control point encapsulates the LCM. The LCM class is a subclass of the control point base class and as such the LCM is able to automatically detect the logical components once they have been deployed in the network. The `ControlPointBase` class hides the UPnP technology from the LCM. The functionality of the PCM expresses itself to an additional CMI interface of the logical component and has been incorporated into the service description of the root device encapsulating the logical component.

3.8.3.2 Logical Component and PCM

The inheritance relation between the logical component class and the `RootDeviceBase` class indicate the UPnP root device encapsulation of the logical component. The logical component and the services it has to offer a described in a description file that is sent to control points. The interfaces the logical component implements can be mapped one-to-one to the services the devices offer to control points. Through the device encapsulation, logical components announce their presence to the LCM on entering the network and through the services of the device the LCM can invoke methods of the interfaces of the logical components.

3.8.4 Proxy / Stub mechanism

A clear separation must be made between the ACM and the UPnP architecture, on which it has been built on, to enable reuse of it and to make the mapping to an embedded system possible. The UPnP technology must easily be discarded when mapped to an embedded system. Figure 19 shows how the proxy / stub mechanism hides the UPnP architecture from the ACM.

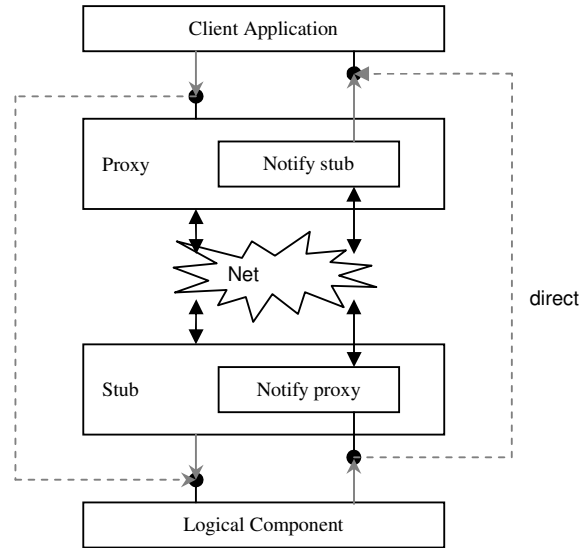


Figure 19: Proxy - Stub mechanism

The proxy – stub mechanism hides the UPnP architecture to the client applications and logical components. Moreover it also hides the (remote) location of the logical components and client application. Each interface of a logical component has an associated proxy. Client applications communicating over an interface are unaware of the presence of the proxy as the proxy provides the exact same interface as provided by the logical component. The same is valid for the notification interfaces provided by the client application. Stubs handle the UPnP actions on the services of the devices encapsulating the logical component and translate the UPnP actions into actual calls on the interface. Each interface of a logical component has an associated notification interface. For this reason each stub contains a notification proxy and each proxy contains a notification stub to handle the notifications.

The proxy – stub mechanism can easily be discarded when ported to an embedded system. Instead of giving the interfaces of the proxies to the client applications the actual interfaces of the logical components can be passed. The proxies and the stubs are simply discarded. This has been depicted in Figure 19 through the dashed arrows that run directly from and to the interface of the logical component and client application respectively.

4 Use case switching

Media processing systems usually implement different configurations and are therefore subject to frequent change of their component structure. Each of these configurations is specified by a use case, see [3.3 Use Case]. This chapter addresses the issues that arise when media processing systems switch from one use case to another. One of these issues is the dynamic reconfiguration of the software component. Often strict requirements are imposed on this reconfiguration process. After identifying and analyzing the issues and requirements that adhere to switching a use case, a strategy for performing this switch is presented.

4.1 Switching conditions

Media processing system often impose strict performance and timing constraints when they switch from one use case to another. Consider for instance the security monitoring system displayed in Figure 20 in which a security guard remotely observes a suspicious person on security camera A. An image enhancer component clarifies he streams from the security camera. When the suspects moves out of range of security camera A, but into security camera B the security guard switches to security camera B. Clearly it is unacceptable that too large a delay is experienced before the stream from security camera B is shown on the screen as within this period the security guard can lose sight of the suspect. The switch should therefore occur in a reasonable amount of time. To do this it must be determined which reconfiguration actions are going to be taken and in which order they are executed.

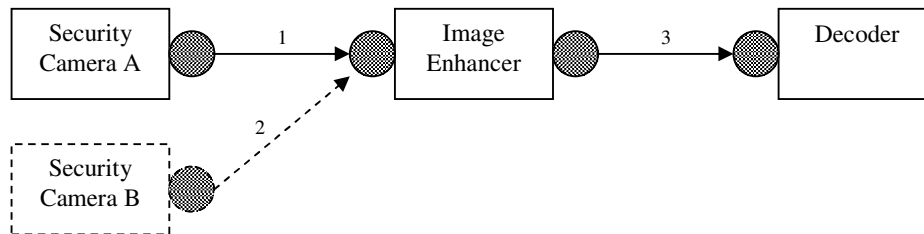


Figure 20: Reconfiguration of component interconnection

A reconfiguration schedule specifies the order in which connections are created and torn down and in which order logical components are started and stopped such that the reconfiguration occurs “smoothly”.

4.2 Dynamic reconfiguration

Reconfiguration is the process of changing the software component composition in which the media processing system is moved from one consistent state to another consistent state. With reconfiguration it is possible to change the behavior and functionality of the media processing system by making different combinations of logical components through different interconnections between them. Reconfiguration entails the gradual change and evolution of one process network to another process network. [Kahn&MacQueen, 1977] even states that a process network can be regarded as starting from one node and gradually expanding through reconfiguration as the computation proceeds. The reconfiguration process must ensure that the

mechanisms of the reconfiguration process gives the correct software component composition and that the reconfiguration maintains correctness of the software component compositions, that no values are lost, introduced or duplicated. [Web et al] refers to this as ensuring the consistency of representation during reconfiguration.

4.2.1 Reconfiguration techniques

Generally reconfiguration techniques can be classified into two classes: static reconfiguration techniques and dynamic reconfiguration techniques.

- Static reconfiguration entails stopping the entire media processing system and then changes the software component composition into the new composition. Once the new software component composition has been created the media processing system can be started again. With static reconfiguration data traffic is stopped and the average packet latency increases dramatically during the reconfiguration. Thus, it will not be possible to guarantee the required timing constraints and “smoothness” constraints. Especially on distributed real-time applications the stopping of the entire media processing system has undesirable effects.
- Dynamic reconfiguration tackles the reconfiguration of the interconnection network from a dynamic point of view. Dynamic reconfiguration performs the reconfiguration process of the software component composition of the media processing system while streams are still flowing through the system.

The kinds of applications that are considered in the STAGE platform are real-time multimedia applications. Distributed real-time multimedia applications have strict timing requirement. Many distributed multi-media applications, such as real-time video compression and decompression, video on-demand servers, distributed databases, etc requires very high network bandwidth and imposes strict requirements when the system is reconfigured. Static reconfiguration techniques are not suitable to perform the use case switch as they impose a too great of latency to meet the timing requirements.

4.2.2 Requirements

In this section the requirements for the dynamic reconfiguration process of the software component composition are described. Basically three requirements are of importance: minimizing the time that is needed for the reconfiguration process, a “smooth” transition from one use case to another and a correct transition.

4.2.2.1 Reconfiguration time

Perhaps the most important requirement for a dynamic reconfiguration strategy is that of shortening the reconfiguration time as best as possible. The reconfiguration time is defined as the time that is needed to change the current composition of logical components to the new composition by changing the interconnections. The reconfiguration time directly relates to the time that elapses from the point the use case switch has been initiated to the point where the switch becomes visible or goes into effect.

4.2.2.2 “Smooth” transition

Dynamic reconfiguration interrupts the streams that are flowing through the media processing system and causes discomfort. Interruptions in streams can result in glitches at the output components due to the lack of streams. The user perceives these glitches as brief moments in which the images on the screens freeze. Another problem that affects the smoothness of transition is that dynamic reconfiguration often leaves residue streams behind in the media processing system. Residue streams are obsolete streams that belonged to the previous configuration. Residue streams affect the behavior of the system and can even break the correctness of the system. They therefore need to be removed.

4.2.2.3 Correctness

Correctness must ensure that a correct reconfiguration process does not leave the software component composition in an inconsistent state. An inconsistent state can for instance be the result of a partial execution of a reconfiguration process in which half way of the reconfiguration process an error occurs. In these situations the media processing system must be brought back to a consistent state; possibly the empty state. Furthermore the reconfiguration process does not semantically break it also important that the reconfiguration process ensures the integrity of the streams, which means that streams. It is important that a logical component is stopped at the correct positions in a stream (for instance only on I-frames in MPEG2 streams). Seen in a broader perspective it is of even importance that a sequence of logical components part of a media processing system are stopped at the same positions in the stream.

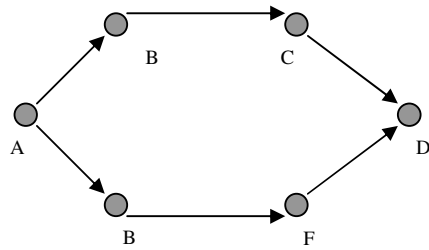
4.3 Dynamic Reconfiguration Strategy

4.3.1 Minimization of reconfiguration time

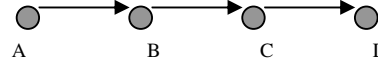
The reconfiguration time of a software component composition is directly related to the following metrics:

- **Number of connections that have to be created and torn down:** The number of connections that have to be created, torn down or switched relates directly to the time that is needed for the dynamic reconfiguration. The less number of connections that need to be created or torn down the shorter the reconfiguration process will take.
- **Reuse of allocated resources:** The employment of new resources is usually a costly operation. It is therefore better to maximize the reuse of the resources that have already been allocated.

Minimizing the reconfiguration time therefore boils down to maximize the reuse of logical components and connection; in other words, maximizing the overlap between the old software component composition and the new composition. The minimization problem can be regarded as a graph problem. Consider for instance the following two graphs.



graph G



graph G'

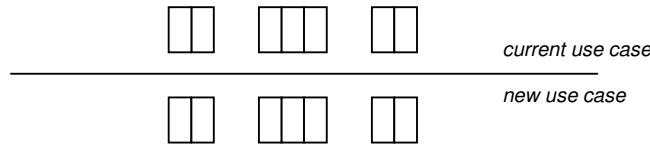
The vertices of the graph represent the logical components and the edges represent the connections between the logical components. Each vertex has a label, which corresponds to the type of the logical component. The minimization of the reconfiguration time now corresponds to minimize the costs inferred with the creation, teardown and relaying of connections. To each of the actions a cost is associated. The problem now consists of finding a mapping from a vertex in G to a vertex in G' with the same label such that the cost associated with it is minimal.

A straightforward solution would be to go to all possible permutations. This is an NP-complete problem as the number of permutation equals $\frac{\max(n,m)!}{(n-m)!}$ and if also taken into account the possibility that vertice can be mapped to an empty vertice the search space grows to $\sum_{i=0}^{\max(n,m)-1} (\max(n-i, m)! / (n-i - m!))$. This problem is better known as X [Had the reference to it, but lost it. Still searching], which describes the problem as NP-complete.

4.3.1.1 Implementation

From the formula given in the previous section it is easily seen that the number of possible mappings grows exponentially compared to the number of vertices in the graph. A complete search into the possible mapping space would therefore be intractable. However, the kind of graphs dealt with within the STAGE platform only contains vertices that vary a great deal in labels. In general at most two or three limits vertices with the same label and in most case all vertices have different labels. It is therefore possible to search through all possible mappings.

Vertices correspond to logical components and labels to the types of the logical components. Logical components in a use case can be mapped to a unique identifier, as it is often desired to map logical components to physical locations. These logical components are excluded from the calculation, because the mapping is fixed and unique. The logical components of the new use case that remain are divided into groups of logical components that have equal types. The same grouping is performed on the logical components of the current use case. Groups of the current and the new use case are places opposite to each other as indicated in the figure below. Empty elements are added to each group to even out the number of elements in it. A mapping of a logical component to an empty value means that resources need to be allocated or freed.



To determine the minimum cost all permutations of the sub groups corresponding to the new use case are run through in a recursive manner. The cost associated with each permutation is calculated by determining the number of connections that need to be created or torn down and the resources that need to be allocated or freed.

4.3.2 Residue Streams

Residue streams can remain in a media processing system after the reconfiguration of it. Residue streams consist of parts of obsolete streams that are possibly scattered throughout the media processing system. Reconsider for instance the situation of the security monitoring system sketched in Figure 20. Here the configuration of the media processing system is changed from viewing the stream from security camera A to security camera B by relaying connection 1 to connection 2. After the reconfiguration process the stream from security camera B must be shown on the screen “immediately”. However due to buffering of streams obsolete streams of security camera A can remain within the system that can disturb the behavior and even break the requirements of a media processing system. In the security monitoring system the effect of residue streams is not that great, because buffers must be kept in order to display the live view with reasonable delay. In certain configurations of media processing systems buffers can however be quite large (for instance playing a stream from the internet) and compositions lengthier. In these situations residue streams have a great impact on the behavior of the system. It is therefore needed that residue streams are removed from the system.

Not every stream in a media processing system becomes obsolete after a reconfiguration process. In general it can be observed that streams only become obsolete when they reside in connections and logical components that can be linked to the reconfiguration point through a connection or a path of connections. In Figure 21 this corresponds with the connections and logical components that are displayed at the right of the reconfiguration points. Reconfiguration points always coincide with output pins of logical components and are the points which are (re)connected in the reconfiguration process. In Figure 20 the reconfiguration point is the output pin of security camera B as this is the pin that is connected in the reconfiguration process. A reconfiguration point that is (in)directly connected to another reconfiguration point cancels the latter reconfiguration point. Figure 21 shows two reconfiguration points. Reconfiguration point 1 encapsulates reconfiguration point 2. The dashed part in the figure shows the parts of the software component composition that need to be flushed for residue streams.

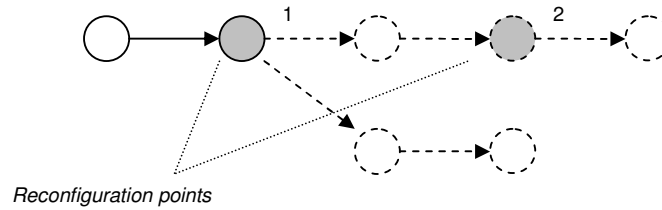


Figure 21: Reconfiguration points

The removal of residue streams is tackled within the ACM through the introduction of a special token that is inserted into the data stream. This token flows throughout the system and passes through tasks unchanged. When this token is encountered by tasks, the task flushes itself and its output ports. Subsequently the special token is written to each of the output ports and the task proceeds as usual. It is essential that only residue streams be removed. The following course of actions is taken by the LCM to remove residue streams and to ensure that only residue streams are removed.

1. The operations of the reconfiguration points are suspended. As a result streams cannot pass this point. The reconfiguration points remain in a suspended state.
2. Once all reconfiguration points have been suspended the dynamic reconfiguration process can be executed.
3. Special tokens are inserted at the reconfiguration points to remove residue streams by flushing the appropriate connections and logical components.
4. Directly after the insertion of the special token the operations of the reconfiguration point are resumed. Streams produced hereafter will not be removed as they have been produced after the insertion of the special token.

The suggested approach is a straightforward and natural way for removing residue streams in a correct manner, but is subjective to optimization. Herein the dynamic reconfiguration process and the removal residue streams are sequential actions. Better smoothness properties can be reached when these actions are executed in parallel. However considerable intelligence has to be present within the LCM as a strict schedule must be followed to correctly remove residue streams.

4.3.3 Conclusion

Designing a strategy for dynamically reconfiguring a media processing system that fulfills the described requirements boils down to reaching some kind of tradeoff between minimizing the reconfiguration time and the degree of “smoothness”. In the perspective of minimizing the reconfiguration time it would be best to maximize the reuse of the existing composition, to stop components as soon as possible irregardless of their position in the stream and to create and teardown connections when possible. However, in the perspective of “smoothness” it is more advantageous to build a separate software component composition next to the existing one and in a single step switch from the old composition to the new. Afterwards the old composition can be cleaned up. This way a “smooth” transition can be achieved.

In practice both extremes are not feasible as inconsistencies in the stream can occur or the resources are simply not available. The adopted dynamic reconfiguration strategy tries to make a trade-off in both requirements while ensuring the correctness of the

reconfiguration. Reconfiguration time is minimized by reusing as much as possible from the existing software component composition thereby avoiding the time-consuming operations of allocating new resources and creating and tearing down connections. Residue streams that remain in the media processing system after reconfiguration are a big threat for “smoothness”. A technique has been described to remove residue streams in a correct manner.

4.4 Reconfiguration and Half Channels

This paragraph will indicate briefly how the half-channel concepts aid in implementing the dynamic reconfiguration process. Consider for instance the situation depicted in Figure 22 which shows three processes that are connected through “regular” channels”. If the process network is reconfigured in which process B disappears and process A and C are connected to each other directly, this would entail the destruction of the both the existing channels and the creation of a new channel that connects processes A and C.

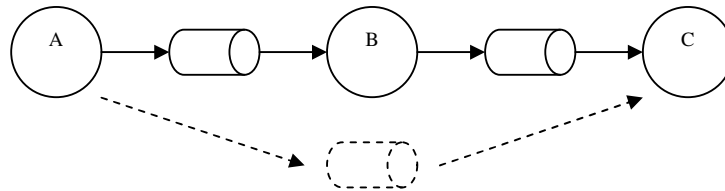


Figure 22: Reconfiguration with "regular" channels

Figure 23 depicts the same situation only this time half channels connect the processes. When the same reconfiguration of the process network is done, processes A and C can be connected by joining the two existing half channels together and destroy the remaining half channels. This reconfiguration operation is cheaper than the previous situation as in this case no new channels need to be created and only two half channels need to be destroyed.

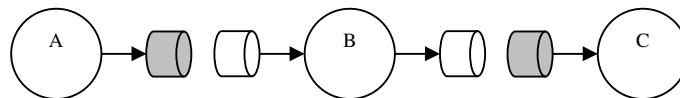


Figure 23: Reconfiguration with half channels

5 RESULTS

5.1 Proof of concept

5.1.1 Setup of demonstrator

The setup of the demonstration that has been used to show and evaluate the working of the ACM is shown in Figure 24. The demonstration runs on 3 PC's. Each of these PC runs the Windows XP operating system. Figure 24 also shows the distribution of the different client applications and components over the different PC's. For clarity sakes each logical component is drawn with only one interface and links that are held by the LCM to each of the connection management interfaces of the logical components have not been drawn. Also omitted in the picture are the notification interfaces that are provided by the client applications and LCM.

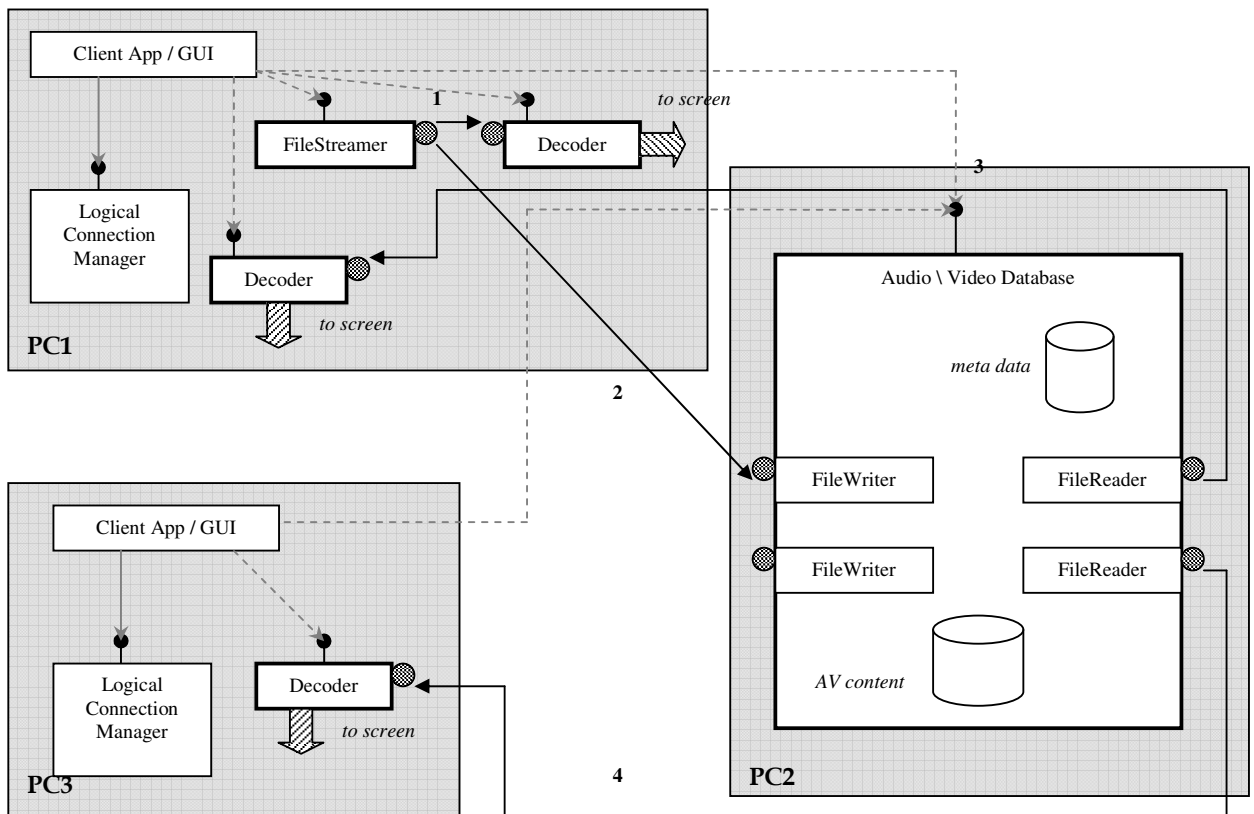


Figure 24: Setup of demonstrator

The logical components in Figure 24 distinguish themselves by a thicker border. Three different logical components are shown. Each of these logical components will be explained in more detail in the next section. Opposed to the dark grey arrows the dashed arrows indicate the interface calls that proceed over the UPnP network and are therefore implemented with proxies and stubs. The proxy / stub mechanism (3.8.4) hide the UPnP architecture and the remote locations of the different entities. Finally logical streaming connections are represented by solid black arrows that run between the pins.

5.1.1.1 Components

The file streamer retrieves streams from files and outputs these onto its output pin. The file streamer has only one output pin and no input pin. Beside the standard connection management interface the file streamer has one additional control interface; the IFileStreaming interface. This interface contains methods for selecting which files to stream from.

The decoder decodes incoming video streams and shows the decoded video stream on the screen. The decoder has one input pin and has a IDecoder interface for control. Typical actions contained in the IDecoder interface are Start, Stop, Pause and Resume. A hardware decoder has been used to decode the MPEG2 streams.

The Audio Video Database (AVDB) is a place for storing and retrieving audio and video content. For this reason the AVDB implements two control interfaces: ITocBrowse and ITocRecPlay. Through the ITocBrowse interface client application can retrieve meta data about the AV content stored at the AVDB. The ITocRecPlay provides methods for recording new video streams and also for playing recorded streams. The AVDB has two input pins for recording new video streams and two output pins for streaming recorded video streams. Each requested recording and play through the ITocRecPlay interface is on a free pin. Which pins are free can be determined through the connection management interface. The AVDB is capable of recording two streams and playing two recorded streams at the same time.

The video streams are simply stored by the AVDB on the file system of the operating system it is running on. Meta data is stored in a MySQL database and to access this database the AVDB connects to a MySQL server.

The demonstrator simulates an in-home network. Each PC represents a different room in the house. PC1 is the parent's room with a television and a set-top box that streams out broadcasts; PC2 is the children's room with just a television and finally the Avdb is placed in the basement of the house. Through the client applications and the GUI the parents and the children can decide what streams to play.

5.2 Implementation details

A logical model gives an overview of the static and structural elements of the ACM and captures the entities and building blocks of it. The logical model of the ACM has been included in appendix 8.2 to provide an overview of the ACM design. This model depicts the important classes, their methods and the basic relationships between the classes.

5.2.1 Graphical User Interface

The graphical user interface (GUI) visualizes the ACM by displaying the logical components that have been plugged in and the logical streaming connections between the pins of the logical components in a graphical way. The visualization is dynamic in the sense that it adapts to the current state of the ACM. Each logical component in the GUI can be in three different states: not plugged in, plugged in and active (part of the current use case). Connections between the pins of logical components appear or disappear in the GUI on the moment they have been created or torn down.

The GUI and the LCM are not part of the same address space, but run in different processes. It has been chosen to do this to get a clear separation of the responsibilities. The GUI in principle merely serves as a utility to provide an intuitive mean to visualize and access the LCM. Stream sockets have been used to communicate over the process boundary. As the communication between the GUI and the LCM is limited by a small set of commands and notifications, the communication straightforwardly consists of the exchange of these commands and notifications

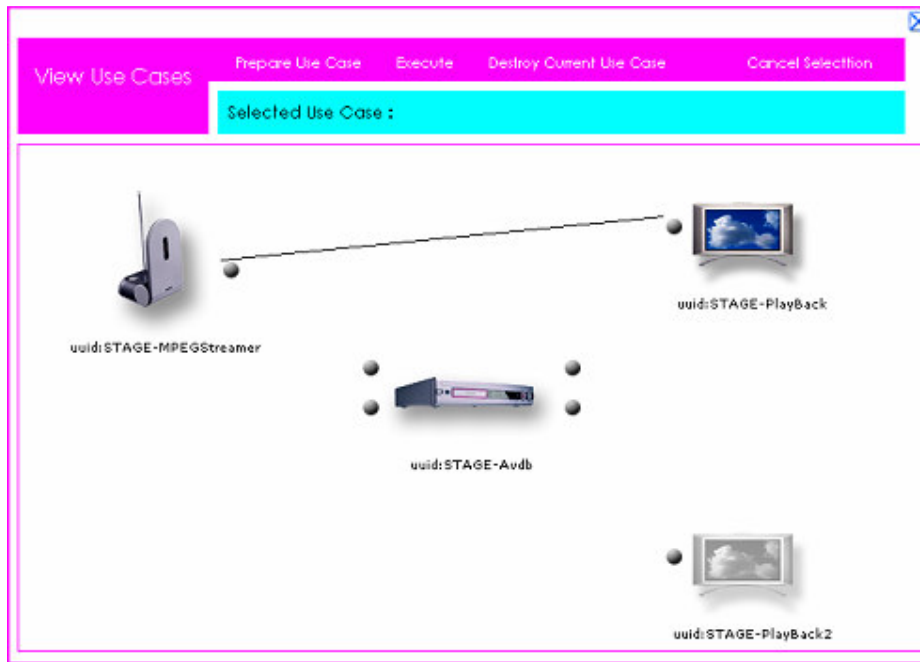


Figure 25: Graphical user interface to visualize ACM

For the demonstrator described in the previous paragraph the following use cases have been identified, implemented and run:

- Live view: show the live broadcast stream that is received by the tuner directly onto the screen. Connection 1 needs to be created
- Record live view: record the live broadcast from the tuner by streaming it to the AVDB. Connection 2 needs to be created.
- Play recorded stream: stream a video stream that had been recorded into the AVDB to a decoder and thereby showing it on the screen. Connection 3 needs to be created.
- Live view while recording live view: combination of showing a live view while in the mean time also record the same live view into the AVDB. Connection 1 and 2 need to be created.
- Play recorded stream while recording live view: Play a stream recorded to the AVDB while at the same time record the live broadcast coming from the tuner. Connection 1 and 3 needs to be setup.
- Play two different recorded streams: play two different recorded streams on two different screens. Connections 3 and 4 need to be setup.

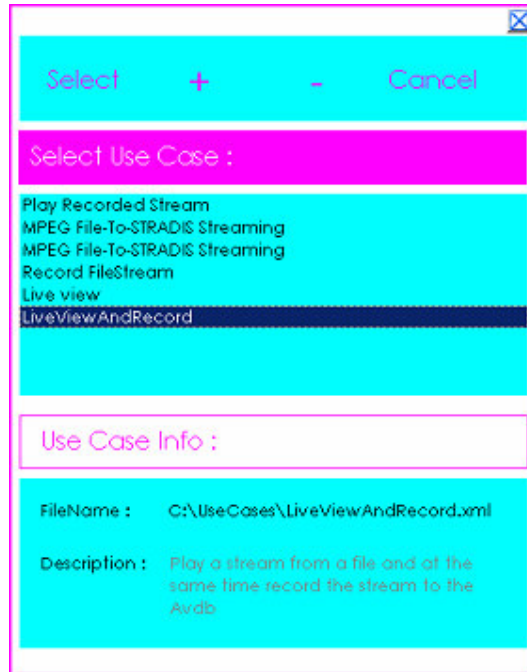


Figure 26: Management of use cases in repository

5.2.2 Media processing system development

The GUI has been developed to provide an environment to provide an intuitive mean for the assessment of product concepts and the development and building of media processing system. Different compositions of logical components can easily and rapidly setup, tried out and evaluated. The proceedings of use case switches can be easily tracked and monitored. The XML description of a use case has been kept minimal without lost of expressiveness. The GUI uses a dedicated directory as a repository for the use cases. Use cases that are added to the GUI are copied to this repository and are automatically loaded at startup of the GUI. Use cases are currently constructed by directly editing the XML description. A true front-end for the construction of use cases is still missing. Such an application could prove to be quite valuable in the assessment of product concepts. An example of an XML description of the “Play a recorded stream” use case has been included in appendix 8.1.

In final media processing systems the different use cases have been determined and fixed and furthermore specific middleware applications and applications as has been depicted in Figure 4 are responsible for the specific control parts of the system. In this situation control invocations are well blended with the setup of the underlying software component composition. The GUI is a generic application that makes no distinction among the use cases. For some use cases control commands need to be invoked on certain control interface before the software component composition is setup. An example of such a use case is the play of a recorded stream. In this use case it is not sufficient to only setup the software component composition, but also it is required to indicate which recorded stream should be played. This requirement should not be stored within the GUI application because it need to be generic, but within the use case description. For this reason the XML description of the use case contains two

additional sections that list the events of which the GUI is notified of prior and after the setup of the software component composition respectively. The GUI therefore not relates control invocation with use cases, but with certain events. This guarantees the independency of the GUI such that is easily extendable and modified. An example of the working of these setup and destroy events is given in the next paragraph.

5.3 Results

5.3.1 Plugging of new logical components

The simple concepts of the UPnP architecture proved to be a valuable asset for the ACM as they provide the means for a dynamic and flexible platform. The UPnP control point and device concept made it possible to plug in logical components to the ACM without extensive configuration. The described setup of demonstration in Figure 24 has been setup in different phases; not only out of practical reasons, but also to determine the steps required for the plug in of a new logical component. The plug in of the AVDB proceeded with great ease. Due to the UPnP architecture, the LCM automatically detected the AVDB logical component as soon the logical component was started. Furthermore as each of the control interfaces of the logical component is identified with an UPnP service description the LCM was also able to discover the services of the logical components. For each of the interfaces of the logical component a proxy factory needs to be registered to the LCM that is able to manufacture these proxies. At this point recompilation of the LCM was inevitable as proxy factories have been linked statically to the LCM. In the future this can be avoided by dynamically link the proxy factories from a dynamic link library (DLL).

Upon startup of the LCM the underlying control point broadcasts a request on reception of which devices send back their advertisements. Devices cannot detect the presence of control points and only announce themselves on startup. The UPnP standard prescribes that devices re-announce themselves every 30 minutes. By broadcast a request on startup the LCM obtains an updated view of the logical components that have been plugged in. For each plugged logical component the LCM obtains a pointer to the connection management interface and subscribes itself to the notifications of this interface.

5.3.2 Reservation policy

The setup of the demonstrator in Figure 24 indicates the presence of two LCM's. Each GUI is accompanied by its own LCM. Perhaps a more favorable approach would be to have a single LCM with which both GUI's interact such that the required intelligence can be concentrated in a single point. The drawback however is that it is a potential single point of failure. The presence of two LCM's can result in conflict when both require the same resources. Although the problems these conflicts are out of the scope of this thesis, a simple reservation policy has been defined and implemented to solve these conflicts. It is certainly not complete or ideal, but is sufficient for the demonstrator.

In this reservation policy each logical component is designated as exclusive or public. Exclusive logical components can only be reserved and controlled by a single client application while multiple client applications can access a public logical component.

In the described demonstrator only the AVDB is designated as public as multiple client applications must be able record streams to and play streams from it. The interfaces of public logical components should not contain any methods that can influence other use cases. Before the LCM sets up a use case it must first reserve the exclusive logical components and determine if the input pins which need to be connected are already connected. If any of these operations fail the use case cannot be setup and already made reservations are released. Connections on output pins that are already connected are allowed and correspond to multi-casts. If the output pin belongs to a reserved logical component, no control interfaces can be obtained by the client application that does not hold the reservation. When the reservation of the logical component is released the LCM must automatically try to reserve it. Deadlock due to race situations cannot occur as the LCM does not wait for a reservation, but simply aborts the setup. Reserved logical components and occupied logical components express themselves in the GUI through different colors.

5.3.3 Flexible software component composition

Software component compositions are setup by the ACM through a generic description of the composition. The generic description enables an easy and rapid way of setting up different software component compositions. From the perspective of product concept assessment this is a crucial fact, as product concept assessment requires an easy and rapid way for trying out different software component compositions. Especially in combination with the GUI the generic description allows for a fast exploration of the possible combinations of the available logical components.

As an example of the setup of a use case the “Play recorded stream” is explained in detail. The “play recorded stream” use case consist of playing a recorded stream that is stored at the AVDB to a decoder. It only consists out of setting up connection 4 in Figure 24. The XML description of the use case is given in appendix 8.1. A sequence diagram is depicted in appendix 8.1 that specifies interaction of the GUI with the LCM for setting up a use case. Upon selection of a use case in the GUI the LCM parses the corresponding XML description. If any logical components or pins have not been specified in the description the LCM maps these to matching entities. As the decoder is an exclusive logical component it needs to be reserved. If the preparation of the use case is successful the use case can be setup. The setup of a use case usually does not only entail the creation of the software component composition, but also requires reconfiguring the behavior of the logical components. The necessary configuration actions have been included in the use case description as setup and destroy events and the LCM notifies the GUI of these events prior to the setup and destruction of the use case respectively. In the “play recorded stream” use case it must be indicated by the GUI which recording needs to be played. A “SetRecordingToPlay” event is sent to the GUI by the LCM and prompts the end-user for specifying the recording to play. Through the `ITocBrowse` interface the user has the ability to get an overview of the available recordings. Once the `recordingId` to be played has been set the LCM can continue setting up the connections to finish the setup of the use case. After completion of the setup the requested recording is directly played and visible on the screen.

For validation of the product concept it is important that the media processing system can be prototyped in real-time on real data sets. The demonstrator indicates that the implementation of the ACM is efficient enough to display the streams that flow through the system fluently. No freezes that are the result of lack of streams occur at the screens if communication is buffered. Streams are sent in chunks of 20 MPEGTS packet sizes at a time. It can be observed that when the size of the chunks is decreased to 1 MPEGTS packet size the components become too fine-grained to allow for a fluent display of the stream. The overhead that is created by file i / o and synchronization then becomes too great.

5.3.4 Evaluation use case switching

Several use cases have been defined as has been indicated in paragraph 5.2.2. In general it can be said the number of possible use cases N equals $N*(N-1)$ as in principle it is possible to switch from one use case to another. Due to the small sizes of the use cases in the demonstrator the focus in the evaluation of use case switching is not on having an optimized result, but on the correct switching between the use cases both in terms of changing the software component composition as in the flow of streams. Furthermore in the demonstrator is also important that when switched between use cases, the resulting change in streams is shown on the displays within a reasonable amount of time. The demonstrator shows that the LCM is able to switch between the use cases correctly.

Some delays in the change of streams on the display have been experienced. For instance when switching from the live view use case to the play of a recorded stream a delay of approximately 3 seconds was experienced before the recorded stream was visible on the display. In this period the old stream kept on playing. Careful analysis shows that this delay was not caused by the ACM, but by the use of an internal buffer (possibly in hardware) in the play back of the stream. More detailed knowledge is required to flush this buffer. The delay was also experienced in the startup of the decoder as the stream was only displayed when the buffer was sufficiently filled.

Furthermore the responsiveness of the ACM in a fully streaming configuration had a noticeable delay from the moment of initiating the use case switch by pressing on a button to the completion of the switch. This difference in delay was especially noticeable when setting up a use case in an idle configuration and setting up or switching a use case in a fully streaming configuration of the media processing system. Invoked control actions on interface needed a considerable amount of time to complete when a reasonable amount of network traffic was present and a number of processes and threads were actively running. Examination of these problems indicates in the direction of the specific UPnP stack implementation, but they could equally as well be caused by the pthread implementation or even the implementation of the ACM.

6 EVALUATION

6.1 Conclusion

This thesis has concentrated on defining the ACM architecture to manage the complex process of developing, building and prototyping of media processing systems. Binding the components of which media processing systems consists in a very late state has proven to be an essential point to rapidly construct media processing systems. Very late binding enables the dynamical setup of different software component compositions.

An important concept within the ACM is logical component. Logical components are abstract notions that encapsulate the functionality of lower-level signal processing functions and expose standardized interfaces to access the implemented functionality. Logical components are the fundamental building blocks of the media processing systems. Pins are the only means for logical components to communicate streaming data with other logical components. By restricting streaming communication on the pins logical components can more easily be “clicked” together and enables a loose coupling between the logical components and the client applications using them.

6.1.1 Development and building of applications

The approach taken by the ACM for building media processing systems is to construct them from implementations of logical components that are running distributed in a PC network. Through this approach media processing systems can be build and prototyped in a rapid manner, which is an essential requirement for the assessment of product concepts. The ACM allows for a development process that deviates from the strict top-down approach and follows a partial bottom-up approach. Media processing systems are not developed and build according to a strict decomposition scheme, but instead the guidelines are roughly sketched and integration proceeds through a more or less trial and error process.

It is important that a software component composition can be defined easily such that different compositions can be rapidly setup. Within the ACM software component compositions are defined in a generic description, so called use case descriptions. The demonstrator shows that the use case descriptions are an excellent mean to specify and prototype different media processing systems as they provide a high-level mean to specify media processing systems. The demonstrator also shows that a GUI provides an intuitive mean for product concept assessment

6.1.2 Dynamic reconfiguration

Media processing systems usually implement more than one use case. An approach has been designed and implemented to automate the switch between the different use cases. This is necessary as the number of possible use case switches grows quadratically with the number of use cases. Use case switching entails the dynamic reconfiguration of a software component composition into another composition without stopping the streams within the media processing system. Two notions are important for the dynamic reconfiguration: the reconfiguration time and the “smoothness” of the transition. Calculating a reconfiguration schedule by maximizing

the reuse of the current software component composition has been used to minimize the reconfiguration time. A dynamic reconfiguration process can leave residue streams in the media processing system. Residue streams can be a threat the correctness of the media processing system and affect the smoothness property. A technique has been described to remove these residue streams and the demonstrator shows that the technique is feasible.

The implemented demonstrator indicates that the designed dynamic reconfiguration approach is feasible. For each possible use case switch the LCM was able to generate a reconfiguration schedule that entailed the least number of logical components and connections to be employed and created respectively. The reconfiguration time needed for the reconfiguration process appeared to be small enough for the switch to cause no obvious irritation. For instance the switch from the live view use case to the play recorded stream use case in the demonstrator occurred smoothly. Due to the small size of the software component compositions the size of the buffers was not relevant. Buffering is introduced for capturing irregularities in processing speed of the tasks and external interventions, i.e. network traffic, workload of processor. These are more likely to occur when the software component compositions are larger.

6.1.3 Using UPnP technology

The UPnP architecture proved to be a valuable supplement to the ACM as it provides the underlying architecture to automatically detect the logical components that are plugged into the prototyping platform. Furthermore the UPnP architecture allows for an easy way of hiding the distributed nature of the media processing system to the client applications. A drawback of using the UPnP architecture is that it adds some overhead to the system as everything occurs through strings. Furthermore it has been noticed that not all implementations of the UPnP stack by the different developers work together flawlessly. Especially the Siemens UPnP stack appeared to have its own specific implementation. For example, each URL (device, control, etc) is automatically prefixed with the unique identifier of the device regardless of the setting of the URLBase. Both the Intel as the Cyberlink implementation of the UPnP stack experienced some problems with this. It has not been investigated if the incompatibilities were the result of the deviation of the UPnP stack implementation from the UPnP standard or the lack of strictness in the UPnP standard.

6.2 Outlook

The emergence of interoperability standards and architectures for applications to use over networks such as the Common Object Request Broker Architecture (CORBA), Distributed COM [COM], UPnP, Web services [Webservices] etc, has led to a platform in which full-blown applications use each others functionality for operation. Furthermore the adoption of a component based development of software has left us clear structured components which are accessed through an abstract set of functionality. A logical step to take is to exploit the functionality that is already made available and exposed by existing software applications. Constructing applications from deployed software applications that run somewhere in the network provides a flexible mean for rapidly create powerful applications with minor effort.

The focus of this thesis has been on applications in the media processing domain and has shown that the complexity of developing such systems benefits from the adopted development approach. However it is not straightforwardly that the same approach can be used in other domains. The nature of the software components of which media processing systems consist lend themselves for this purpose as they can be easily “clicked” together in a pipe-line fashion. In other domains the couplings between the software components are much tighter and complex. Loose coupling and a high-level abstraction are key requirements for the composability and reuse of software components. So although the flexible way in which applications can be created seems to be promising, further investigations and research is needed in order to determine if the same approach is applicable to software systems outside the media processing domain. The Service Oriented Architecture (SOA) heads into this direction. SOA is a distributed software model in which software applications are built from services on a network that communicate with each other. Services are defined at an abstract level such as business processes. The idea of services is to decouple the data and the processing of it.

6.3 Future Work

6.3.1 Error Handling

The logical components that are part of a distributed media processing system do not share any global state. Error handling is therefore hard to address. Care has to be taken that an error handling design does not mix errors from the logical components that are executing in parallel in an unpredictable way. The error handling strategy must bring the media processing system to a consistent state in case of error like a crash. In many situations the entire media processing system has to be reinitialized. At least it must be prevented that the whole media processing system must be restarted. If the media processing system proceeds in some incremental way resynchronization is a possibility. An option to do is to introduce special markers in the data stream to resynchronize the logical components and which pass unchanged through the entire media processing system. The media processing system can then be reinitialized at the stage at which the failure occurred. The design of the error handling in the ACM is not straightforward, but must be designed with care.

6.3.2 Design Environment

The ACM provides a back-end for the flexible and dynamical construction of media processing systems. These applications or use cases are constructed from generic description files that have been denoted in XML. Use cases are currently specified by manually editing these description files. Although the implemented graphical user interface visualizes the internal workings of the ACM and provides a shallow approach for designing media processing systems, a more advanced (graphical) environment is required to design and construct media processing systems. In this environment media processing systems can be created by dragging and dropping logical components that might currently deployed in the network and defines the connections between them. The use case description can form the bridge between the design environment and the LCM as has been indicated in Figure 27. An analogy can be made to the Rapid Application Development environments such as Borland Delphi in which applications are created by drag-and-drop of visual components.

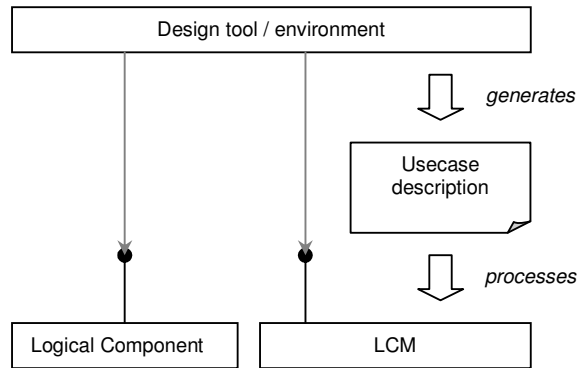


Figure 27: Design tool

The design tool in principle emulates the media processing system. The construction of a media processing system does not merely entail the setup of the underlying software component composition, but also requires application specific control to change the behavior of the logical components. Care must be taken when designing the design environment that it is not tailored to application specifics as it needs to remain a generic environment. Currently this has been implemented by incorporating setup and destroys events in the use case descriptions that are recognized and performed by the GUI. This approach avoids knowledge within the GUI of dependencies between certain events and use cases.

6.3.3 Quality-of-Service (QoS)

The ACM could be extended with QoS concepts to prioritize the use of the resources. Fundamentally QoS enables a better provision of service, like dedicated bandwidth, controlled latency and loss characteristics. For instance in some situations it is permitted to lose parts of streams. QoS enables better control over resources resulting in prioritizing the certain flows and a more efficient of the resources. Tailored services can be offered to applications that match their needs. For instance for mission-critical applications more resources are employed than for applications that are not critical. Currently the ACM has minor control over the amount of resources employed for a certain component.

7 REFERENCES

- [Chan et al, 2002] Chan, S et al. – *ConnectionManager:1 Service Template Version 1.01 for Universal Plug and Play Version 1.0* – June 25, 2002 – Microsoft Corporation
- [COM] Component Object Model [COM],
<http://www.microsoft.com/com/tech/com.asp>
Distributed Component Object Model [DCOM],
<http://www.microsoft.com/com/tech/DCOM.asp>
- [CORBA] <http://www.corba.org>
- [Gamma et al, 2000] Gamma, E et al – *Design Patterns, de Nederlandse editie, Elementen van herbruikbare objectgeoriënteerde software* – Addison Wesley Professional – 2000.
- [Gstreamer] <http://gstreamer.freedesktop.org/>
- [Heineman & Council, 2001] G.Heineman and W.Council - *Component-Based Software Engineering / Putting the pieces together* – Addison Wesley, 2001 – pages 33-48
- [IntelDeviceSpy] Device Spy: Intel's Universal Control Point
<http://www.intel.com/technology/UPnP/tech.htm#tools>
- [JavaBeans] <http://java.sun.com/products/javabeans/>
- [Kahn, 1974] G.Kahn – *The semantics of a simple language for parallel programming* – Information Processing, J.L.Rosenfeld(ed.) – North-Holland Publishing Co, 1974
- [Kahn&MacQueen, 1977] G.Kahn & D.B. MacQueen – *Coroutines and Networks of Parallel Processes* – In B. Gilchrist, editor - Proceedings of IFIP Congress 77, pages 993-998 – North Holland Publishing, 1977
- [Lange & Kang, 2004] A.A.J.deLange & I.C.Kang – *The STAGE platform, Architecture Prototyping of Consumer Systems with PC networks* – March 2004 – Philips Research Laboratories – Eindhoven, The Netherlands.

- [Ledeczi et al, 2000] A.Ledeczi, G.Karsi, T.Bapty – *Synthesis of Self-Adaptive Software* – Institute for Software Integrated Systems, Vanderbilt University, Nashville – In the proceedings of the IEEE Aerospace 2000 Conference.
- [Mitchell et al, 1998] S.Mitchell, H.Ngauib, G.Coulouris, T.Kindelberg – *Dynamically Reconfiguring Multimedia Components: A Model-based Approach* – Department of Computer Science, Queen Mary and Westfield College, University of London – 1998
- [Nierstrasz & Meijler, 1995] O.Nierstrasz and T.D.Meijler – *Research Directions in Software Composition* – Software Composition Group, University of Berne – In ACM Computing Surveys, vol 27, no. 2, June 1995, pp. 262-264.
- [Parks, 1995] T.Parks – *Bounded scheduling of process networks* – Ph.D. dissertation, University of California at Berkeley, 1995.
- [Rothermel et al, 1994] K.Rothermel, I.Barth, T.Helbig – *Cinema – An Architecture for Distributed Multimedia Applications* – University of Stuttgart – In Architecture and Protocols for High Speed Networks, pp. 253-271, Kluwer Academic Publishers, 1994.
- [Schmidth et al, 2000] D.Schmidth et al – *Pattern-oriented software architecture, vol 2, patterns for concurrent and networked objects* – John Wiley and sons, 2002
- [SOA] T. Datz – *What you need to know about Service Oriented Architecture* – <http://www.cio.com/archive/011504/soa.html>
- [Szyperski, 1997] C.Szyperski – *Component Software, Beyond Object-Oriented Programming* – Addison Wesley 1997
- [Tanenbaum, 2002] A.Tanenbaum & M.v.Steen – *Distributed systems: principles and paradigms* – Prentice Hall, 2002
- [UPnP, 2003] UPnP Forum – *UPnPtm Device Architecture 1.0* – Version 1.0.1, 6 May 2003
- [UPnPSiemens] Siemens Stack C++ for UPnP technologies – <http://www.plugin-play-technologies.com/downloads.html>

[Vayssiere et al, 1999]	J.Vayssiere, D.Webb, A.Wendelborn – Distributed Process Networks – Department of Computer Science, University of Adelaide, Australia – October 28, 1999.
[Web et al]	Webb, D & Wendelborn, A.L. & Vayssiere, J – <i>A Study of Computational Reconfiguration in a Process Network</i> – Department of Computer Science, University of Adelaide – South Australia 5005, Australia
{Webservices}	Web Services Architecture, http://www.w3.org/TR/2003/WD-ws-arch-20030808/
[XML, 2000]	E.R.Harold – <i>XML, het complete handboek</i> – Academic Service, Schoonhoven, 2000

8 APPENDICES

8.1 Use Case XML Description File

The following XML use case description is an example of a use case in which a recorded stream is played from the Avdb and displayed on a screen.

```
<?xml version="1.0"?>
<useCase>
  <generalInfo>
    <name>PlayRecordedStream</name>
    <description> Play a recorded stream from the Avdb </description>
  </generalInfo>

  <componentComposition>
    <logicalComponentListing>
      <logicalComponent>
        <id>uuid:STAGE-Avdb</id>
        <type>urn:schemas-upnp-org:device:Avdb:1</type>
        <meta></meta>
      </logicalComponent>
      <logicalComponent>
        <id>uuid:STAGE-PlayBack</id>
        <type>urn:schemas-upnp-org:device:Decoder:1</type>
        <meta></meta>
      </logicalComponent>
    </logicalComponentListing>

    <connectionListing>
      <connection>
        <from>
          <id>uuid:STAGE-Avdb</id>
          <pin>3</pin>
        </from>
        <to>
          <id>uuid:STAGE-PlayBack</id>
          <pin>1</pin>
        </to>
        <protocol>HalfChannel</protocol>
        <priority>1</priority>
        <capacity>10</capacity>
        <meta></meta>
      </connection>
    </connectionListing>
  </componentComposition>
  <onSetup>
    <event>
      <eventId>SetPlayRecordedStream</eventId>
      <param>3</param>
    </event>
  </onSetup>
  <onDestroy></onDestroy>
</useCase>
```

Explanation of the different sections of the XML specifications:

- **<generalInfo> .. </generalInfo>**
This section gives some general information about the use case like the name of the use case and a short description of the use case.
- **<componentComposition> .. </componentComposition>**
Specification of the software component composition that is associated with each use case. The software component composition is specified by listing the logical components that are required and the connections between the pins of the logical components.
- **<logicalComponent> .. </logicalComponent>**
A logical component is specified by an optional unique identifier of the logical component and a required type definition. If a logical component is not accompanied by a unique identifier, but instead has been left unspecified, an instance of a logical component with the same type will be mapped to it by the LCM during the setup of the use case. The meta section of the logical component contains meta-information about the logical component. A specification tool can for instance require this meta-information to function properly.
- **<connection> .. </connection>**
A connection always runs between an output pin of one logical component to the input pin of another logical component. The **<from>** and **<to>** section specifies the logical components and the pins between which the connection runs. The protocol section specifies which mechanism is used for transporting the streaming data over the connection. The capacity section specifies the capacity of the connection. Each connection can be assigned a priority. This priority influences the order in which the connections are created during the building of the software component composition. The lower the number of the priority, the higher the priority of the connection.
- **<onSetup> .. </onSetup>, <onDestroy> .. onDestroy>**
The events that are listed in these sections are sent to the client application that has requested for the setup of the use case during the setup and destruction of the use case respectively. The proper setup of a use case does not only entail the setup of the interconnection between the logical components, but is also determined by application specific parameters that go beyond the connection manager. Which parameters need to be set by the application are specified in these sections

8.2 Logical Model

The logical model describes the static structural elements of the ACM and captures the entities and building blocks of it. Class elements in the logical model are the design elements and components are built from classes at run-time. The logical model provides a high-level view of the ACM and defines the basic relationships between the different entities.

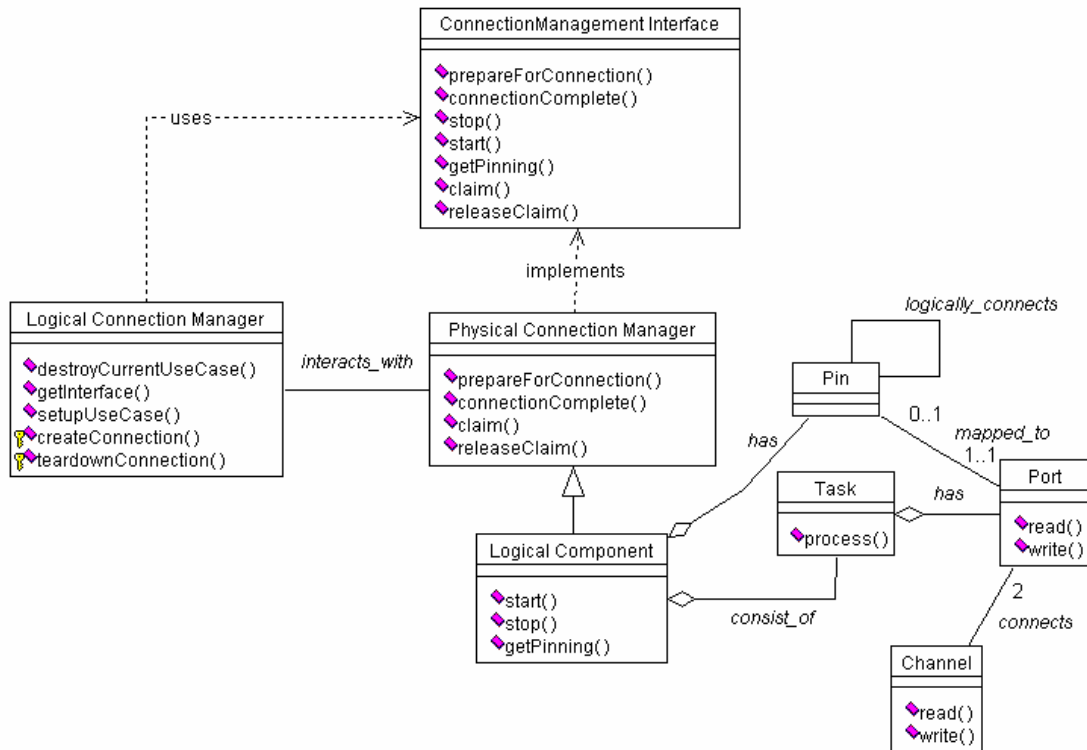


Figure 28: Use case diagram of important classes

The Logical Component class is an abstract class and all specific logical components, like the AVDB and Decoder logical component as depicted in Figure 24, are derived from this class. The logical component class implements the common functionality of the logical component and is a derived class of the RootDeviceBase class to incorporate the functionality of the UPnP device. The mechanisms and working of the RootDeviceBase class is hidden by the logical component class from the specific logical components. Each logical component has zero or more pins that can be logically connected to each other. Each pin of a logical component is uniquely mapped to a port of a task, but not the other way around. Pins are essentially the ports of tasks of which logical components comprise that are made public at the logical component level.

Just as the logical component the task class is the parent class for all specific tasks. Examples of these derived tasks are the specific tasks of the MPEG2 decoder in Figure 6. The process loop of a task actively pulls and pushes data from and to its port thereby transforming the data in between. The processing loop of the tasks runs in a

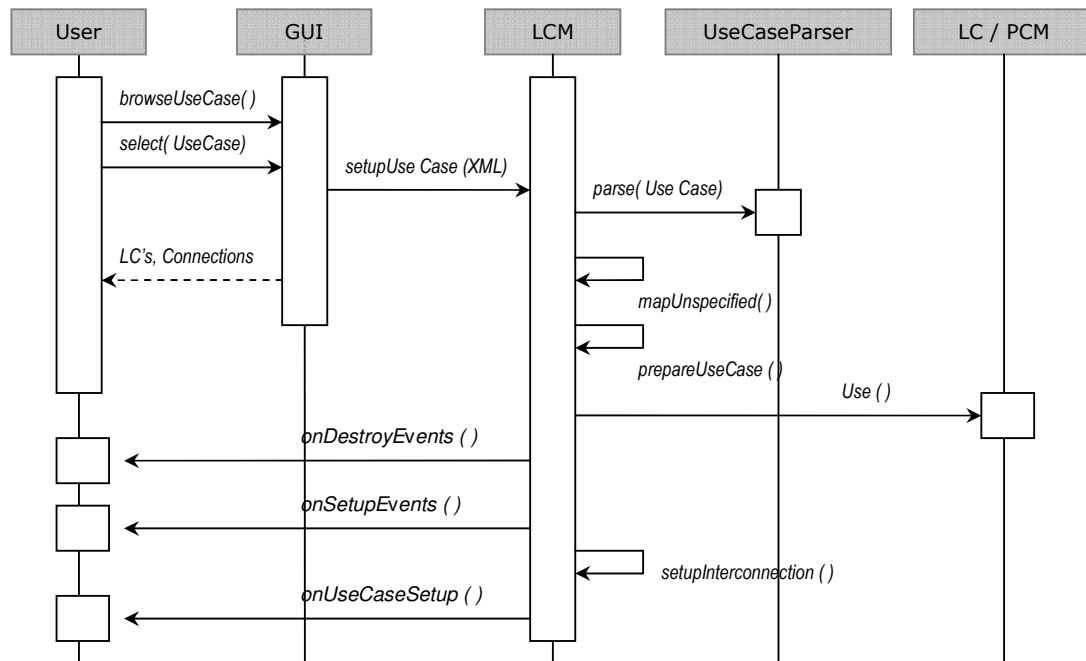
separate thread. The task class takes care of the proper implementation and handling of the threads and maintains state-information of the task. Each tasks has one or more input and output ports. Ports are connected to each other by means of channels. Channels abstract from the physical communication and transportation medium and are accessed through a simple read and write interface. Both the logical component and the task class hide the details of the ACM to enable developers to concentrate on the technology they need to develop and implement.

The LCM is a deployable software entity that provides methods for creating software component compositions by creating the connections between the pins of logical components. Connections between logical components are created by the LCM through the connection management interface that is provided by the PCM, which is associated with each logical component. The logical component is a subclass from the PCM. Through the connection management interface the LCM not only sets up connections, but also retrieves information about the logical component. Each logical component in the platform is detected by the LCM through the UPnP control point and root device encapsulation of the LCM and logical component respectively.

8.3 Sequence Diagrams

8.3.1 Use Case Setup

Sequence Diagram 1 shows the sequence diagram for setting up a use case. It is assumed that the XML description of the use case is correct and that the use case can be setup successfully.



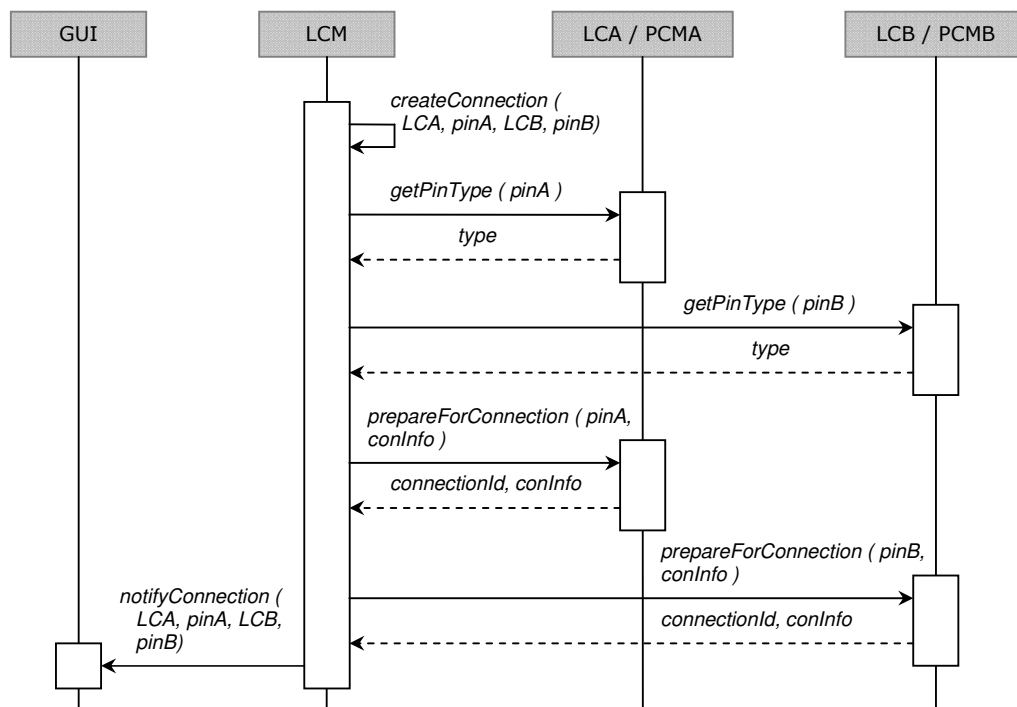
Sequence Diagram 1: Setup of use case

The setup of use case starts with an end-user browsing through the defined use cases and selecting one of them. Upon selection of a use case the XML description of the use case is passed on to the LCM for preparation. The use case is subsequently parsed

by the UseCaseParser. Each use case can contain references to unspecified logical components or pins. In these situations the LCM needs to map and match these logical components that have been plugged in to the ACM and to free pins of logical components. In the `prepareUseCase()` method it is determined if the software component composition can actually be setup by among other things checking if required logical components have been plugged in and if so are reserved.

Once the use case has been prepared successfully it can be setup. Before the software component composition is setup, the LCM notifies the GUI of the destroy events of the current use case and subsequently of the setup events of the prepared use case. These events give the application the ability to deal with application specific configuration of the logical components during the destruction and setup of a use case respectively. After these events the current software component composition is dynamically reconfigured by the LCM into the new software component composition by changing the interconnections and flushing the appropriate connections and components. When the use case has been setup successfully by the LCM the GUI is notified of this fact.

8.4 Creation of connection



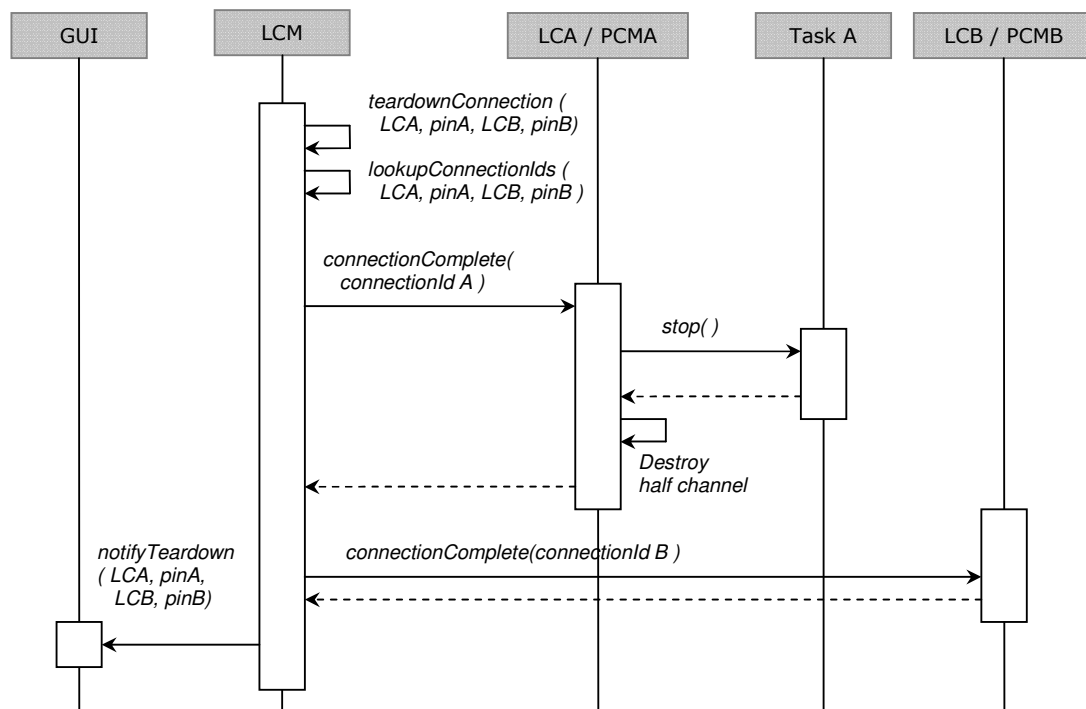
Sequence Diagram 2: Creation of connection

On execution of a use case the LCM sets up the software component composition by creating the appropriate connections between the pins of logical components. Before a connection can be created the LCM checks if the types of the pins of the logical components are equal. If this is the case the connection is prepared at pin_A of LCA through the connection management interface implemented by the PCM that is associated with each logical component. The PCMA allocates internal resources that

are necessary for the connection and links. The PCMA returns an internally unique identifier for the connection together with information about the prepared connection back to the LCM. The returned connection information is used as an argument for the `prepareForConnection()` invocation at LCB. Also at this side the resources for the connection are allocated and linked. PCMB returns an internally unique identifier together with information about the connection at LCB to the LCM. Sequence Diagram 2 does not show the details at LCB / PCMB as these are symmetric to the operations at LCA / PCMA.

A connection is identified by two connection identifiers. Although individually the identifiers are not unique the combination of the two connection identifiers is unique. Every communication of the LCM with the PCM about a connection proceeds from here on through this connection identifier. If a connection has been created successfully by the LCM a notification of the creation is sent to the GUI.

8.5 Teardown of connection



Sequence Diagram 3: Teardown of connection

A connection needs to be torn down when the connection is no longer part of the new use case (possibly none). The teardown of a connection between the pins of logical components begins with looking up the identifiers to which the connection is known to both PCM's. Once the connection identifiers have been looked up the `connectionComplete()` method is invoked to tear down the connection at LCA / PCMA. Before the connection is torn down the task operating on the half channel, which has been mapped to the connection, is stopped. The half channel is then destroyed by PCMA. Subsequently the LCM also tears down the connection at PCMB.

A notification is sent out by the LCM if the connection has been successfully torn down.

8.6 Task Interaction

Tasks interact with each other through their ports. A channel is formed by connecting two ports together. By using explicit named channels a loose coupling between the tasks can be reached. From the point of view of task A it is not important which task consumes the data it is producing; it does not need to know. The same is valid when seen from the point of view of task B.

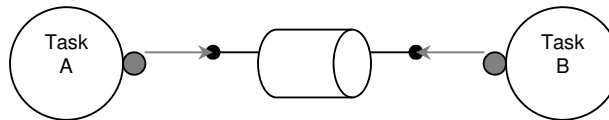
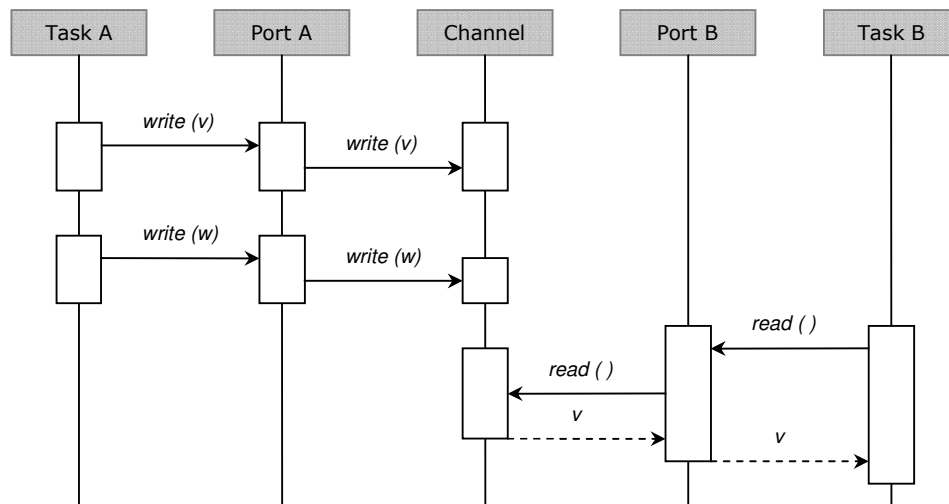


Figure 29: Interacting tasks

Communication between the tasks is persistent and proceeds in an asynchronous manner [Tanenbaum, 2002]. This means that a message that has been produced by task A is stored by the channel as long as it takes for task B to consume it. In Sequence Diagram 4 it can be seen that a message v sent by task A is stored by the channel until task B consumes it even though task A has produced a message w in the mean time.



Sequence Diagram 4: Interaction between tasks

Sent messages are buffered by the channel and allows task A to continue immediately after submitting the message for transmission. Producing messages to channels that are full results in a block until a message is consumed from the channel. Channels that have zero capacity are equal to synchronous communication.

8.6.1 Half-Channel Specification

Each channel c is sliced into two half-channels ($hc1$, $hc2$). $c!$ and $c?$ denote a write and read action on the channel respectively.

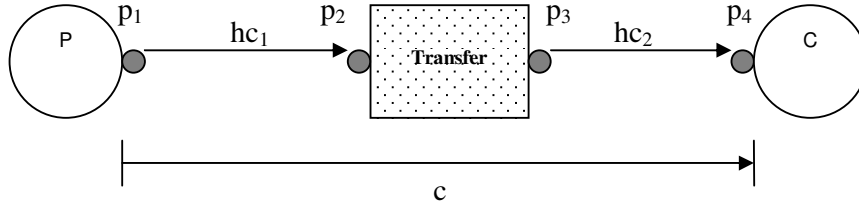


Figure 30: Specification Half-Channel

Channel c is defined by the pair of ports (p_1, p_4) . The producer process P writes values to port p_1 , while the consumer process reads from port p_4 . A special process Transfer is introduced to connect the two half-channels (hc_1, hc_2) of channel c . A channel is unidirectional and reliable. The following predicates are defined to indicate the connectedness of the channel:

- $\Theta(p_1)$: port p_1 is connected to by a half channel to another port.
- $\Theta(p_1, p_2)$: port p_1 is connected to by a half channel to port p_2 .
- $\Xi(p_1, p_2)$: port p_1 is connected to by a channel to port p_2 .

The buffering of the half channels is made explicit by defining the following notation:

Buffers: $hc.b, E$

With each half-channel a buffer is associated. A special value E indicates the empty buffer. For the specification of the half-channel the buffer is regarded as a list on which the following operators are defined:

- $[] ++ x$: Appends element x to the list.
- $\text{tail}([])$: Removes the head of the list.
- $\text{head}([])$: Returns the head of the list.

Three basic operations are defined : $\text{read}(\text{port}, x)$ for writing the value of x to the port, $\text{read}(\text{port}, x)$ for reading a value to x from the port and $\text{transfer}(\text{port}_1, \text{port}_2)$ for reading a value from port_1 and subsequently writing it to port_2 . The semantics of these operations are specified in more detail in the subsections.

8.6.1.1 Properties

Boundedness

- P0 : $0 \leq \#write(p_1) - \#transfer(p_2, p_3) \leq N$
- P1 : $0 \leq \#transfer(p_2, p_3) - \#read(p_4) \leq M$
- P2 : $0 \leq \#write(p_1) - \#read(p_4) \leq N$

Half channel hc_2 is a mirror of hc_1 . Therefore the upper bound of P2 is N and not $N+M$. Values that are contained in hc_2 are also contained in hc_1 and values that are in hc_1 , but not in hc_2 will eventually be transferred to hc_2 .

- P3: $\langle \forall v: (v \in hc_2) \Rightarrow (v \in hc_1) \rangle$

- P4: $\langle \forall v: ((v \in hc_1) \wedge (v \notin hc_2)) \Rightarrow \diamond(v \in hc_2) \rangle$,
where \diamond indicates eventually

Blocking on the channel operations only occur when P2 is violated, i.e. the buffer associated with the channel is empty or full.

FIFO property

- P5: $\langle \forall x,y: ((write(p_1, x) < write(p_1, y)) \Rightarrow (read(p_4, x) < read(p_4, y))) \rangle$
where $<$ indicates the order in which the action was executed. If value x is written to the channel before value y then also value x is read before value y .

Operations

- write(p_1, x) : await (($hc_1.b < N$) \wedge ($\Theta(p_1)$));
 $hc_1.b := hc_1.b ++ x$;
- read(p_4, x) : await(($hc_2.b \neq E$) \wedge ($\Theta(p_4)$);
 $hc_2.b, hc_1.b, x := tail(hc_2.b), tail(hc_1.b), head(hc_2.b)$
- transfer(p_2, p_3) : await($\Theta(p_2) \wedge \Theta(p_3)$);
 await(($hc_1.b \neq E$) \wedge ($hc_2.b < M$));
 $hc_2.b := hc_2.b ++ head(hc_1.b)$
- flush(hc_1) : $hc_1.b, hc_2.b := E, E$;
- flush(hc_2) : $hc_1.b, hc_2.b := E, E$;

Figure 31 shows the relay of channel (p_1, p_4) to (p_1, p_5)

- relay(p_1, p_4, p_5) : { $\Theta(p_1) \wedge \Theta(p_4) \wedge \neg\Theta(p_5) \wedge \mathcal{E}(p_1, p_4)$ }
 destroy(hc_2) : $hc_2.b = E$;
 { $\Theta(p_1) \wedge \neg\Theta(p_4) \wedge \neg\Theta(p_5) \wedge \neg\mathcal{E}(p_1, p_4)$ }
 create(hc_3): $hc_3.b = E$;
 { $\Theta(p_1) \wedge \neg\Theta(p_4) \wedge \Theta(p_5) \wedge \mathcal{E}(p_1, p_5)$ }

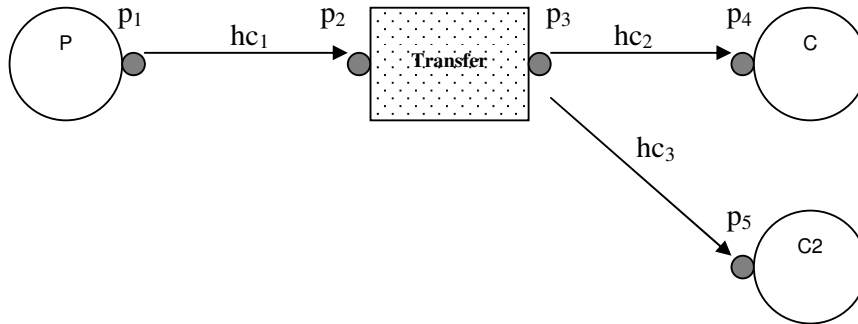


Figure 31: Relay of half-channels

The destruction of half channel hc_2 has no influence on the connectedness of half channel hc_1 and therefore producer process P can keep on writing to port p_1 . The half channel hc_3 upon creation is empty. Due to P4 all values contained in hc_1 will eventually be transferred to the buffer of hc_3 .

8.6.1.2 Implementation

The half-channel notion has been implemented by means of queues (Figure 32). Each half-channel of a channel holds its own queue. The union of the queues of both the half-channels represents the content of the entire channel.

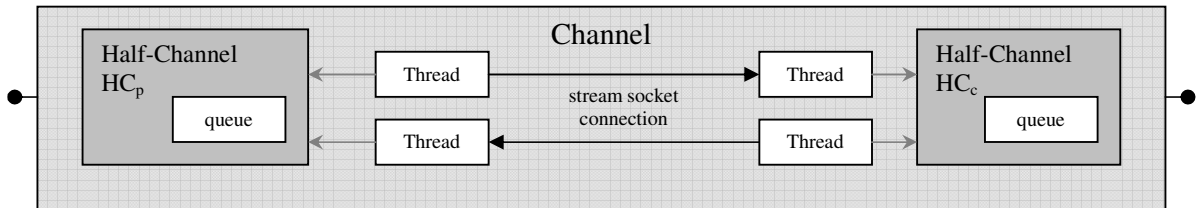


Figure 32: Half-Channel implementation

Synchronization threads are responsible for keeping the queues of the half-channels consistent. Each half-channel has associated with it two synchronization threads: one thread for sending synchronization signals to the associated half-channel and one thread for receiving synchronization signals from the other thread. An example of the two most important synchronization signals is the following:

- Write action on HC_p : the token written to HC_p is transferred to HC_c , but the space occupied by this token is not released yet.
- Read on HC_c : a synchronization signal of the consumption is sent to HC_p and the space occupied by this token in the queue is released.

Half-channels hide the latency introduced by the remote location of processes as much as possible by mirroring the queue at both locations and by keeping the queues synchronized. Each process therefore performs local read and writes. Synchronization occurs in the background. The hiding of latency will be minimal or not at all when the processes are very fine-grained.

8.6.1.3 Results

For evaluating the latency hiding of the half channel the setup displayed in Figure 33 has been used. The solid arrows represent the channels between the ports.

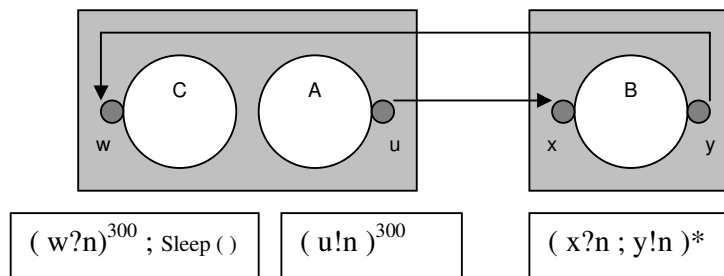


Figure 33: Test setup; feedback loop

Task A and C run on one PC and task B on another. The network consists of a 10 Mbit Ethernet connection. The measured time indicates the time that is needed for the reception of 300 tokens at task C where each token is ten times the size of a MPEG2TS packet (3*4096). The measured time is the elapsed time between the first write of a token by task A and the reception of the 300th token at task C. To simulate the processing times of the tasks, task C reads token at fixed intervals. The capacity of the channels has been set to ten tokens. The following results have been obtained when the half channel is compared to the direct channel in which the channel is allocated at the producer and as such the consumers perform remote invocation to obtain the tokens.

Read interval (ms)	Direct Channel (ms)	Half Channel (ms)	Difference (ms)
0	67327	68108	-781
100	69721	68238	1483
200	87696	68289	19407
400	147352	120854	26498
800	267395	241026	26369

From the results it is apparent that the advantages of the half-channels are the most when the processing time of task C becomes larger. The greater the read intervals the more synchronized the half-channels are. The efficiency gained by the half-channel is the result that both the producer and consumer perform local calls on the channel while the direct channel requires remote procedure calls from the consumer. It can be seen that when the tasks are too fine-grained the benefit of the half-channel is little as the synchronization between the half-channels creates overhead. Two interesting things can be observed:

1. The required time does not change much for read intervals of 0 and 100 ms. This is caused by the fact that a 0 ms sleep time is not actually equal to a zero ms sleep time due to thread switches and switching to processes that are active in the background. At 200 ms the required time still remains the same for the half channel. From this point the working of the half channel becomes visible as cost that is associated with the remote procedure calls is hidden by the half channel. The overhead time introduced by thread and processes switches is therefore somewhere between 100 and 200 ms per read.
2. The difference becomes stable when the read interval becomes too great. The half channel in these cases is fully synchronized and no more efficiency can be gained. The total number of tokens that are communicated remains constant.