Multimedia Framework for Augmented Reality Applications in Ubiquitous Environments

Andrej van der Zee

Waseda University
Department of Information and Computer Science
School of Science and Engineering
Supervisor: Prof. Tatsuo Nakajima

University of Amsterdam
Natural Science, Mathematics and Computer Science
Supervisor: Prof. Dr. Paul Klint

Spring, 2003

Andrej van der Zee
Oosterpark 79
1092AV Amsterdam
The Netherlands
Phone : +31-(0)6-54391387
Fax : +31-(0)20-6652273
E-mail : mavdzee@yahoo.co.uk

Acknowledgements

During my studies in Computer Science at the University of Amsterdam I became interested in the topic of *ubiquitous computing*. Basically, ubiquitous computing is a futuristic vision whereby physical and cyber spaces are seamlessly integrated, providing support to users in the performance of their daily tasks. The subject is still in its initial stages and therefore mainly studied at various research departments. However, although ubiquitous environments are far from commercially ready, the subject is gaining popularity around the world, especially in Japan.

In recognition of this I decided to write my master's thesis on the subject of ubiquitous computing. But where could I find a supporting and stimulating environment for this? My quest led me to Waseda University, Tokyo, a prestigious university well known in Japan and Asia. At the Distributed Computing Laboratory (DCL) at Waseda University, founded in 1999, ubiquitous computing is the main topic of research. After lodging my application, I was finally accepted as a member of the DCL from April 2002 until March 2003. The result of the research is this thesis, which is used for my graduation paper for my master's degree in Computer Science at the University of Amsterdam.

There are two people to whom I owe great gratitude for the realization of this master's thesis. First, my supervisor at the University of Amsterdam, Prof. Dr. Paul Klint, for his kindness and helpfulness in bringing me onto the right track. Second, my supervisor at Waseda University, Prof. Tatsuo Nakajima, for having a *gaikokujin* join his research department and his support in developing and exploring this field of study; also, I would like to thank him for giving me the valuable experience of presenting the research work at the International Conference on Real-Time Embedded Computing Systems and Applications in Tainain, Taiwan, in February 2003.

In addition, I want to thank all my Japanese colleagues at the DCL for their inspiring conversations, my family for their infinite support and, last but not least, my girlfriend Miho Tanaka for surpassing my wildest expectations about my stay in Japan.

Finally, I owe a special word to the financial contributors that made it possible to live and work in Japan for the last year: Waseda University Scholarship, Stichting Bekker la Bastide Fonds, Schuurman Schimmel van Outeren Stichting, Fundatie van de Vrijvrouwe van Renswoude te 's-Gravenhage, and Stichting Dr. Hendrik Muller's Vaderlandsch Fonds.

Andrej van der Zee
Amsterdam/Tokyo, Spring 2003

Contents

Acknowledgements i

List of Figures vii

1 Introduction 1

1.1 Ubiquitous Computing 1

1.2 Augmented Reality for Ubiquitous Environments 1

1.3 Problems of Overall Research Project 3

1.4 Proposed Middleware 3

1.5 Assignment and Research Question 4

1.6 Structure 4

2 Architecture 7

2.1 Multimedia Components 7

2.1.1 CORBA Interface 8

2.1.2 Multimedia Objects 8

2.2 Data Streams 9

2.2.1 Basic Streams 10

2.2.2 Cyclic Streams 10

2.2.3 Broadcasting Streams 10

2.2.4 Distributed Streams 10

2.2.5 Competing Streams 11

2.3 Multimedia Data 11

3 Design and Implementation 13

3.1 Framework Size 13

3.2 Multimedia Components 14

3.2.1 CORBA Interface 14

3.2.2 Multimedia Objects 17

3.2.3 Remote Method Invocation . . . 21
3.3 Object Interaction . . . 22
3.3.1 Communication . . . 22
3.3.2 Synchronization . . . 23
3.4 Data Streams . . . 23
3.4.1 Basic Streams . . . 24
3.4.2 Cyclic Streams . . . 24
3.4.3 Broadcasting Streams . . . 25
3.4.4 Distributed Streams . . . 25
3.4.5 Competing Streams . . . 27
3.5 Multimedia Data . . . 28

4 Usage **31**
4.1 Object Implementation . . . 31
4.2 Component Composition . . . 32
4.3 Component Execution . . . 33
4.4 Stream Configuration . . . 33

5 Framework in Operation **35**
5.1 Augmented Reality Class Library . . . 35
5.1.1 Extended Data Type . . . 35
5.1.2 Detectors . . . 35
5.1.3 Sensors . . . 36
5.1.4 Renderers . . . 36
5.2 Applications . . . 37
5.2.1 Follow-Me Application . . . 37
5.2.2 Mobile Augmented Reality . . . 38

6 Evaluation **39**
6.1 Data Discard . . . 39
6.2 Stream Priority . . . 40
6.3 Distributed Augmented Reality . . . 40

7 Discussion **43**
7.1 Strengths . . . 43
7.2 Weaknesses . . . 44
7.3 Future Work . . . 44

8 Summary **45**

A CORBA Basics **47**
A.1 Intoduction 47
A.2 Naming Service 48
A.3 Example 48

B Object Execution **51**
B.1 Sources 51
B.2 Filters 52
B.3 Sinks 53

Index 55

Bibliography 57

List of Figures

1.1 Augmented Reality. 2
1.2 Middleware architecture. 3

2.1 General component. 7
2.2 Example component. 7
2.3 A source, filter and sink object. 9
2.4 Basic stream. 10
2.5 Cyclic stream. 10
2.6 Broadcasting stream. 10
2.7 Distributed stream. 10
2.8 Competing streams. 11

3.1 UML class diagram for multimedia components. 14
3.2 UML class diagram for CORBA interface. 15
3.3 UML class diagram for multimedia objects. 18
3.4 Object execution for sources, filters and sinks. 20
3.5 UML sequence diagram for remote method invocation. 21
3.6 UML collaboration diagram for object interaction. 22
3.7 Stream tables for a basic stream. 24
3.8 Stream tables for a cyclic stream. 24
3.9 Stream tables for a broadcasting stream. 25
3.10 UML class diagram for TCP servers and clients. 27
3.11 Stream tables for a distributed stream. 27
3.12 UML class diagram for multimedia data. 29

5.1 Augmented reality class library. 36
5.2 Augmented reality application. 37
5.3 Mobile augmented reality application. 38
5.4 PDA view. 38

6.1 Data discard for increasing consuming time. 39
6.2 Competing streams. 40
6.3 Received items for varying priority stream b. 40
6.4 Three distributed streams. 41
6.5 Processing time for 2000 frames. 41

A.1 Subset of Object Request Broker. 47

Chapter 1

Introduction

Augmented reality is an important technology for the realization of ubiquitous environments. The complexities inherent to such environments make it difficult to develop augmented reality applications. The DCL research department proposes middleware to tackle these difficulties in order to simplify the development of such applications.

1.1 Ubiquitous Computing

In the vision of *ubiquitous* or *pervasive computing*[15, 16, 27], physical spaces such as home and office environments will be augmented with numerous integrated devices. User interaction with such environment becomes more seamless and assists users perform their daily activities. Various sensors capture contextual information and make the environments smart. For example, location sensors attached to objects or persons identify and record the location of the object or person, and sound recorders analyze the mood of a person. Such dynamic contextual information conveyed by sensors is used by ubiquitous software to give the user the right service at the right time.

Further examples of applications for ubiquitous environments could be a mobile user interface for controlling appliances in a home environment where the interface is shown on the nearest display to the user. Future home environments have numerous interaction devices such as desktop computers, televisions, PDAs, cellular phones and game controllers connected to a network. The application logic combines dynamic contextual information (user location) with static contextual information (device location and characteristics) from data repositories to reflect context change: as the user moves through the physical space, the nearest input and output device is selected. A simplified version of this example application is built by the DCL research department (Section 5.2.1).

1.2 Augmented Reality for Ubiquitous Environments

One of the purposes of ubiquitous computing is the integration of the real world with cyber space. One technique for achieving this objective is *augmented reality*[13, 14] whereby the human-computer interaction becomes more seamless. Traditionally, users instruct computers explicitly to perform a certain task. Using augmented reality, users will be able to interact

with the real world, which is augmented with computer-generated information to facilitate performing their daily tasks. Augmented reality differs from virtual reality where users interact with an environment that is computer-generated as a whole.

Augmented reality is realized by superimposing digital information on video images capturing the real world. Objects in the real world contain visual markers that are recognized by the augmented reality software and superimposed by computer synthetic information. The resulting images are shown on a display (see Figure 1.1). The software uses contextual information in order to provide a context-specific augmentation. This technique enables the user to interact with the real world instead of with the computer directly. Several freely available libraries for rapid development of augmented reality applications already exist such as the ARToolkit[13] and TRIP[11].

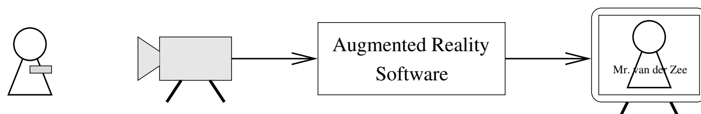

Figure 1.1: Augmented Reality.

A practical example of this can be had by considering the following application of augmented reality in a ubiquitous environment. A student stands in front of the bulletin board in the entrance hall of his university. He takes his PDA equipped with a camera in his hand and directs the camera to the bulletin board. The board has an attached visual marker that is captured by the camera. The PDA streams the video images together with the student's ID to a nearby high-performance computer that runs the augmented reality application. The application performs video analysis on the received images and recognizes the visual marker. It accesses a database and finds today's schedule for that student. The schedule is superimposed on the video images and streamed back to the PDA for display. Consequently, the student sees the bulletin board augmented with his schedule for that day on its display. In this example, the contextual information used for the generation of ad-hoc information is the current time and the student ID.

Consider another example of an augmented reality application in a ubiquitous environment. A refrigerator with an attached visual marker is captured by a nearby camera. The camera sends the images to a high-performance machine running the augmented reality application. The application recognizes the marker and superimposes the images with textual information about the number of bottles inside the refrigerator. When a user with a cellular phone moves to a place nearby the refrigerator, the superimposed images are sent to the cellular phone for display. Consequently, the user sees how many bottles are inside without opening the refrigerator. In this example, the contextual information is user's location and the number of bottles inside the refrigerator. This example application is built by the DCL research department (Section 5.2.2).

Other research departments that study augmented reality in ubiquitous environments use wearable hardware to make the interaction with the augmented real world more attractive. For example, the UbiCom[6] research program at Delft University of Technology uses a wearable terminal and a lightweight see-through display. In the display the user can see virtual information that augments reality, projected over and properly integrated with the real world. The wearable system contains a radio link that connects the user to the ubiquitous computing resources and the Internet. A camera captures the user's environment. Camera images are sent to the backbone and matched to a 3-D description of the real world to determine the user's position and to answer questions of the user that relate to the environment.

1.3 Problems of Overall Research Project

Complexities inherent in ubiquitous environments make building applications very difficult. Ubiquitous environments contain numerous low- and high-performance appliances connected to a network. Portable devices such as PDAs and cellular phones are not powerful enough for expensive augmented reality computation, but they are important candidates for the realization of ubiquitous environments using augmented reality features. Therefore, it is necessary to decompose and distribute applications such that heavy computation is assigned to high-performance machines.

Besides the distribution of applications, augmented reality software has to provide a context-specific augmentation of the real world. Augmented reality in ubiquitous environments facilitates the integration of physical and cyber space, aiming for the assistance of its users in the performance of their daily activities. Therefore, context-specific augmentations are necessary and applications have to become context-aware.

Distribution and context-awareness require considerable effort and skill from the developer. Middleware is needed to tackle these complexities. Our research department proposes such middleware. Its objective is to simplify and accelerate the development of augmented reality applications in ubiquitous environments.

1.4 Proposed Middleware

The DCL research department proposes a middleware based on the Common Object Request Broker Architecture (CORBA)[29, 30] depicted in Figure 1.2. The middleware consists of three parts, which will be explained briefly: the *multimedia framework*, the *communication infrastructure* and the *application composer*.

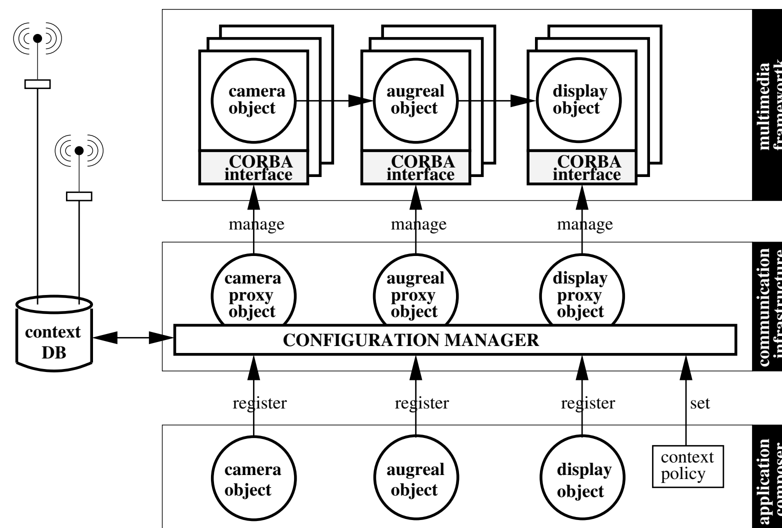

Figure 1.2: Middleware architecture.

The multimedia framework is used for building multimedia components. A component is composed of multimedia objects and a CORBA interface. Clients configure streams between

multimedia objects through the CORBA interface. For example, in Figure 1.2, a data stream is configured from a camera object to a display object through an augmented reality object. Typically, in ubiquitous environments multiple components providing the same service run on different computers at the same time. Consequently, the application has a choice between different components providing identical services.

The CORBA-based communication infrastructure consists of a *configuration manager* that manages *proxy objects*. A proxy object holds a reference to a multimedia object and can be updated by the 'most appropriate' reference according to current context. The configuration manager has access to the *context database* that stores dynamic and static contextual information such as what multimedia components are running, device characteristics and user location. The manager uses this information to determine the most appropriate object reference held by its registered proxy objects and updates the references when needed. In addition, when a reference managed by a proxy object is updated all the affected data streams are reconfigured by the communication infrastructure.

The application composer coordinates the entire application. Developers create proxy objects specifying services rather than explicit object references and register the proxy objects with the configuration manager. Developers configure data streams between multimedia objects and specify policies to control automatic reconfiguration in response to context change. For example, a developer might define a *location policy* as "use service S nearest to object A" or "use service S in host Y or Z" and a *performance policy* as "use service S on 'light' loaded host" or "use service S on any host". Because the references managed by proxy objects are automatically updated by the communication infrastructure according to context policy, the application always uses the most appropriate object transparently.

1.5 Assignment and Research Question

The DCL research group is concerned with the realization of the middleware outlined in the last section. The objective of the middleware is the simplification of the development of augmented reality applications in ubiquitous environments. My task in the overall project is the design and implementation of the multimedia framework.

This paper focuses mainly on my contribution and motivates design decisions and tradeoffs, but also discusses how the framework fits into the overall picture. Moreover, this paper attempts to answer in which extend this particular solution, with the emphasis on the multimedia framework, realizes its objective.

1.6 Structure

This master's thesis discusses the multimedia framework that is part of a middleware that aims for fast and easy development of augmented reality applications in ubiquitous environments. This introduction outlines the need for such a software infrastructure, describes how the multimedia framework fits into the broader scheme of things and states the research question this paper attempts to answer.

The middleware described in the introduction imposes certain requirements on the multimedia framework. Additionally, the objective of the middleware, fast and easy development of augmented reality ubiquitous applications, makes necessary a comprehensive and easy to use multimedia framework. Chapter 2 describes the requirements and the architecture of the

framework. The actual design and implementation, and the decisions and tradeoffs involved, are discussed in Chapter 3.

The multimedia framework is designed for fast and easy usage while maintaining flexibility and extensibility. How the framework is used to build multimedia components and objects is the topic of Chapter 4. Using the framework, the DCL research department developed an augmented reality class library and two sample applications using the library. Both, the library and the sample applications, are described in Chapter 5.

The middleware and the multimedia framework are evaluated by performing tests on custom applications and analyzing the results. This evaluation is the subject of Chapter 6, while the discussion is covered in Chapter 7 and the paper is concluded in Chapter 8.

Chapter 2

Architecture

Ubiquitous environments contain many kinds of devices connected in a network where low-performance computers such as PDAs and cellular phones are too slow for augmented reality processing. In order to run augmented reality applications on such low-performance devices the applications must be decomposed and distributed in such a way that heavy computation is assigned to more powerful computers.

To help the developer with the decomposition and distribution of augmented reality applications, a component-based multimedia framework was decided on. Developers build applications by composing *multimedia components* and configuring data streams within and between components.

This chapter reveals the characteristics of multimedia components and pictures how multimedia data is streamed within or between components in more detail.

2.1 Multimedia Components

A multimedia component is composed of a *CORBA interface*[1] and *multimedia objects* (Figure 2.1). Clients configure streams between multimedia objects by invoking the appropriate operations in the CORBA interface. For example, a component might contain three objects: a camera object for capturing images of the real world, an augmented reality object for superimposing digital images at specific locations within a video frame, and a display object for showing video images on the screen. Through the CORBA interface, a stream can be configured as depicted in Figure 2.2.

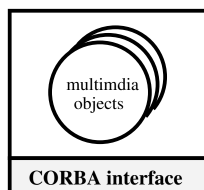

Figure 2.1: General component.

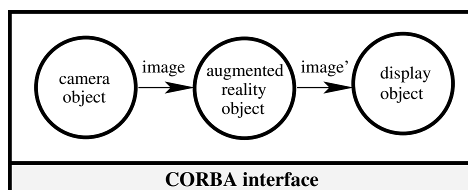

Figure 2.2: Example component.

[1]CORBA is explained briefly in Appendix A.

Components provide *services* to other components. Services are implemented by means of multimedia objects. Components can utilize services provided by other components by streaming multimedia data to the object implementing the service, possibly receiving the manipulated data for further processing afterwards.

Multimedia components are self-describing entities and register themselves at the CORBA Naming Service under a user-specified name. Clients can query the Naming Service for available components and obtain CORBA object references to registered components from the Naming Service.

2.1.1 CORBA Interface

A multimedia component can be remotely accessed through its CORBA interface. After a CORBA component registers itself at the Naming Service, a client retrieves a CORBA object reference to the component, in the remainder of this paper called *CORBA component reference* for brevity, by specifying the component's name. A client uses the CORBA component reference in order to:

- query for the characteristics of the component as a whole, such as component's name and the multimedia objects it contains, or
- query for the characteristics of an individual multimedia object, such as object type and stream information, or
- configure an individual multimedia object for a data stream, or
- create and connect specialized objects for configuring a distributed stream between two objects contained by remote components, or
- change the state of an individual multimedia object contained by the component, such as resolution or frame size of a camera object.

The CORBA interface is identical for all components except for changing the state of individual multimedia objects. Object state is dependent on the characteristics of that object resulting in different CORBA interfaces. The identical part is referred to as the *standard CORBA interface*.

2.1.2 Multimedia Objects

Components contain multimedia objects that implement specific services for other components. Services or multimedia objects need only to be developed once and can be reused in any component. New components are developed with little effort by composing them from existing objects. The framework distinguishes three types of objects which will be introduced next.

Multimedia data is streamed from data producers to data consumers through data manipulators, similar to the VuSystem[5]. Data producers typically are controllers for video or audio capture hardware or media storage hardware. In this paper they are called *sources*. Data manipulators perform operations on the multimedia data that flows through them. Data manipulators get their data from sources or other data manipulators and stream the

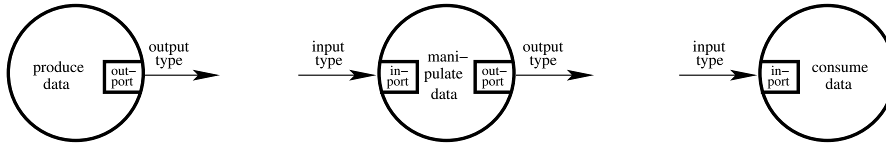

Figure 2.3: A source, filter and sink object.

modified data to a consumer or another manipulator. Here they are called *filters*. Data consumers are multimedia objects that eventually consume the data. Data consumers typically are controllers for media playback or storage devices. In this paper they are called *sinks*.

Furthermore, a multimedia object is categorized as an *input* and/or *output* object. An input object has an *inport* for managing a single buffer for each inflowing data stream and has specified *input type*. An output object has an *outport* for managing stream information for the redirection of outflowing data and has a specified *output type*. For example, a filter is both an input and an output object, meaning it is capable of respectively receiving and sending data. Clearly, a source is an output-only and a sink is an input-only object. See Figure 2.3 for the different categories of multimedia objects this framework distinguishes.

Within component scope, an object is uniquely identified by its *object identifier*. The object identifier is used to access one specific object within a component. Within the global scope, a tuple consisting of a CORBA component reference and an object identifier is used to specify one specific object. Such tuples are called *universal object identifiers*. For example, a client may access one specific object by invoking an operation on the CORBA component reference, passing the object identifier as an input parameter.

All multimedia objects run in separate threads. Priority values are assigned as a criteria for preemption as multiple threads are competing for the CPU simultaneously. In this way, the underlying operating system decides which thread utilizes most CPU cycles during execution. For example, a developer of a multimedia component may assign higher priorities to multimedia objects that perform important calculations. Furthermore, a multimedia object is not scheduled for the CPU until it has received data in one of its input buffers. Consequently, data items function as scheduling tokens for object execution and idle objects do not waist any CPU cycles.

2.2 Data Streams

The central concept of this framework is the streaming of data between multimedia objects. A data stream has one beginning and one or more end points. The beginning is represented by a source that produces the data and the ending by a sink that consumes the data. Usually, a stream contains one or more filters between the data producer and data consumer. For two consecutive objects in the stream, the input and output type need to match. The standard CORBA interface provides functionality for the configuration of data streams.

Four types of streams that need support from the framework are distinguished: *basic*, *cyclic*, *broadcasting* and *distributed streams*. These types can be combined to configure more complex stream types. Next, each type of stream will be construed, followed by an introduction to competing streams.

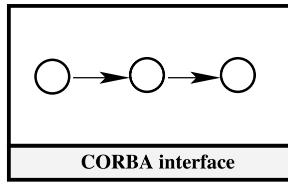

Figure 2.4: Basic stream.

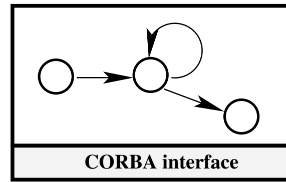

Figure 2.5: Cyclic stream.

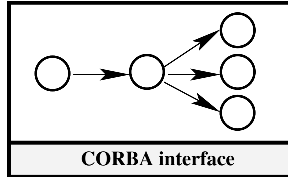

Figure 2.6: Broadcasting stream.

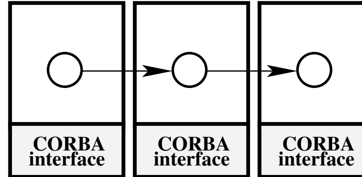

Figure 2.7: Distributed stream.

2.2.1 Basic Streams

In a basic stream, multimedia data flows from a source to a sink through zero or more different filters. Figure 2.4 shows an example of a basic stream with one filter. Basic stream types are the most elementary.

2.2.2 Cyclic Streams

A cyclic stream is a basic stream where the data flows more than one time through one of its participating filters. Figure 2.5 shows an example of a cyclic stream. Before data reaches the sink, it flows two times through the filter.

2.2.3 Broadcasting Streams

In a broadcasting stream, multimedia data is multiplied in one of its output objects and broadcasted to more than one input object. Figure 2.6 illustrates how a filter multiplies its inflowing data and broadcasts the data to three different sinks. Alternatively, three basic streams could be configured through the same filter, resulting in three copies of the same data being identically processed by the filter. Clearly, broadcasting eliminates redundant data processing resulting in more efficient data streams.

2.2.4 Distributed Streams

In a distributed stream, multimedia data flows through objects belonging to different components, possibly running on remote machines. Data is streamed between the components before it reaches remote objects. Figure 2.7 illustrates a distributed stream, where all the objects are contained by different components. Such streams are required for the distribution of augmented reality applications.

2.2.5 Competing Streams

When a multimedia object utilizes a service provided by a filter, it sends its multimedia data to the filter first and receives the manipulated data for further processing afterwards. The providing object exports its service to a theoretically unlimited number of utilizing objects. This implies that a filter requires the ability to be configurable for multiple streams at the same time.

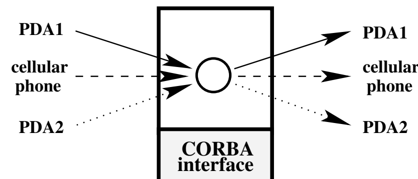

Figure 2.8: Competing streams.

For example, a typical augmented reality filter might analyze video frames for the detection of visual markers and add marker information to the video data structure. Its input type might be video data and its output type might be video data with marker information. Such augmented reality objects perform heavy calculations and for that reason would be executed on a high-performance machine. Low-performance clients, such as PDAs and cellular phones, might utilize the augmented reality service by configuring a data stream as depicted in Figure 2.8.

As multiple streams might compete for the same service at the same time, clients may want to distinguish more important from less important streams. Therefore clients are allowed to assign priority values to streams. In this approach, streams with higher priority values have higher throughput.

2.3 Multimedia Data

Multimedia data is produced by a source and afterwards forwarded to objects next in the stream until the data arrives in its configured sink that consumes the data. When two consecutive objects in a stream belong to remote components, data is transmitted between the components before it reaches its target object. Therefore data needs support for the encoding into a byte stream and decoding from a byte stream. Clearly, encoding and decoding are inverse functions (i.e. `decode(encode(d)) = d` for all data items `d`).

Another issue involved when dealing with streams between objects is the potential of overflowing buffers. Consider a basic stream that consists of one source and one sink and the source generates 10 data items per second. Consequently, approximately every 100 milliseconds one data item arrives in the inport belonging to the sink. If the sink consumes one data item in more then 100 milliseconds, it simply can not keep up with the inflowing data rate. Consequently, its input buffer will overflow causing the container component to crash. More generally, if an input object receives data items at a faster rate than can be processed, its buffers will overflow and execution will be aborted.

When dealing with real-time media, the above issue of buffer overflow causes another side effect. Suppose we have infinite resources and buffers can grow for ever. Now consider the example component in Figure 2.2 where the camera object captures one video frame every 100

milliseconds. If the augmented reality object processes video frames in periods of longer than 100 milliseconds, the superimposed images can never arrive in time for playback. Besides the increasing memory consumption of the executing component, the result is playback of obsolete video frames in slow-motion. An easier way to understand this is to realize that the original time difference between two consecutive captured video frames is 100 milliseconds. But after the frames have been processed by the augmented reality object, the time difference is increased to the processing time of one video frame.

Discarding obsolete data prevents buffers from overflowing and assures playback of up-to-date frames at the price of data-loss. An input object can decide whether a data item is obsolete by comparing the original time difference between the processed item and the next candidate in its input buffer with the processing time of the last item, i.e. the time difference between the last and current fetch. If the former is smaller than the latter, the candidate is obsolete and therefore to be discarded. The next data item in the input buffer is checked until the condition is satisfied. Clearly, to record the original time difference between two consecutive data items, multimedia data requires timestamp support from the framework.

Another requirement is related to broadcasting streams. Before data is broadcast, it must be multiplied. Therefore multimedia data needs cloning support from our framework. In addition, another important issue is generality. New data types might be added to the framework or developers might add their own custom data types. Therefore structure of the multimedia data should be designed in a general and extensible way.

Chapter 3

Design and Implementation

The multimedia framework, described in the previous chapter, is designed for easy usage and fast development of multimedia components and objects. For this purpose, abstractions are defined as C++ classes that are to be extended by developers. How the abstractions are realized is the topic of Section 3.2.

Typical component configurations contain multiple objects executed in separate threads that stream multimedia data between them. Section 3.3 unravels how objects contained by one component communicate and synchronize for the exchange of data.

The central concept of the multimedia framework is the streaming of data. In an application, developers configure streams between multimedia objects, possibly contained by remote components. Data streams are the subject of Section 3.4.

Multimedia data requires basic support from the framework for correct streaming. In addition, developers might want to add more complex, custom data types. How the framework addresses these issues is clarified in Section 3.5.

3.1 Framework Size

The multimedia framework consists of CORBA IDL[1] definitions of the component's interface and C++ classes for the implementation. To give the reader an idea about the size of the framework, the following metrics were found:

- **CORBA IDL.** The IDL code counts 83 lines, 4 files, 1 module, 3 interfaces, 15 operations and 4 exceptions.

- **C++.** The C++ code consists of 25 classes implemented in 2943 lines of code. Excluding (copy-) constructors, the classes make up for 134 methods of which 38 are virtual, 6 are static and 3 are template methods.

[1] CORBA IDL is briefly explained in A.

3.2 Multimedia Components

The C++ class `MComponent` implements an abstraction of a multimedia component (Figure 3.1[2]). The class is composed of one CORBA interface, implemented by `MInterface`, and zero or more multimedia objects, derivations of `MObject`.

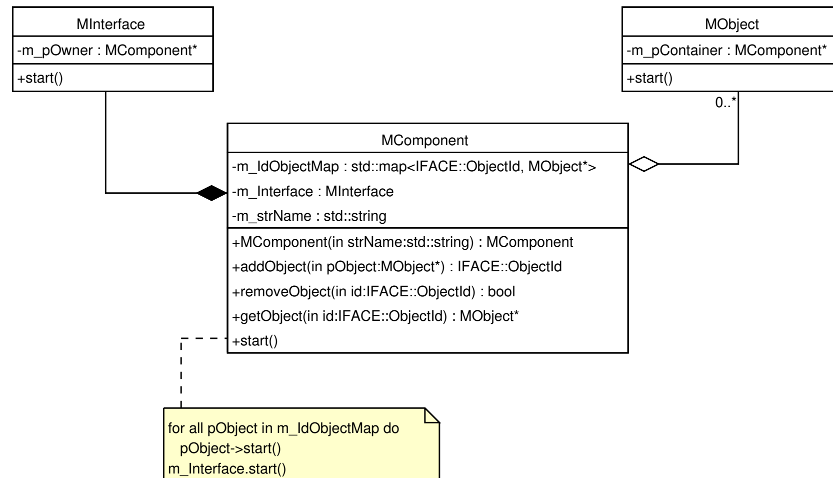

Figure 3.1: UML class diagram for multimedia components.

Many CORBA related issues, such as object incarnation and registration at the CORBA Naming Service, are identical for all components. There is no reason for the component developer to re-invent the wheel. Instead, the `MInterface` class hides most CORBA related complexities so developers can focus on object implementation and component composition instead.

A developer composes a component by deriving from `MComponent` and adding multimedia objects through the `addObject` method. Internally, the component assigns a unique object identifier to the object and stores the pair `[ObjectId, MObject*]` in a STL-map data structure. A requirement is that objects can be remotely accessed through the CORBA interface. Therefore, the framework deals with object identifiers rather than C++ pointers to the objects. C++ pointers are not valid outside the process where they are created and consequently cannot be used in remote components.

A developer starts the component by invoking `start` on an instance of the class. As a result, all objects contained by the component, and the component's interface, are executed in separate threads. The interface registers the component at the CORBA Naming Service under the name passed as an argument to the `MComponent` constructor. After registration, the component is up and running, waiting for incoming CORBA requests.

3.2.1 CORBA Interface

In Section 2.1.1 the CORBA interface was described to be partially identical for all components. As component developers are responsible for designing and implementing the variable

[2] All classes that are part of the framework are preceded by an M for multimedia. In addition, the class diagrams are strongly reduced for clarity.

part, the CORBA interface is partitioned into three parts: the *component interface*, the *stream interface* and the *state interface*. The stream interface is identical for all components and provides functionality for stream configuration. The state interface is component specific depending on the multimedia objects it contains and therefore the responsibility of the developer. The component interface is primarily added to provide one single CORBA object reference to identify one component. The component interface is the implementation of the CORBA component reference introduced in Section 2.1.1.

All interfaces are part of the module `IFACE` written in CORBA IDL[3, 4]. The IDL interface definitions are implemented by C++ classes with corresponding names with an `_i` suffix as depicted in the UML class diagram in Figure 3.2. Referring to the diagram, `MInterface` is a composition of the three classes implementing the CORBA interface. It hides CORBA related issues such as activation of a POA manager, incarnation of CORBA objects representing the three interfaces and registration of its component at the CORBA Naming Service under a user-specified name. Following, the three CORBA interfaces are listed and briefly explained individually.

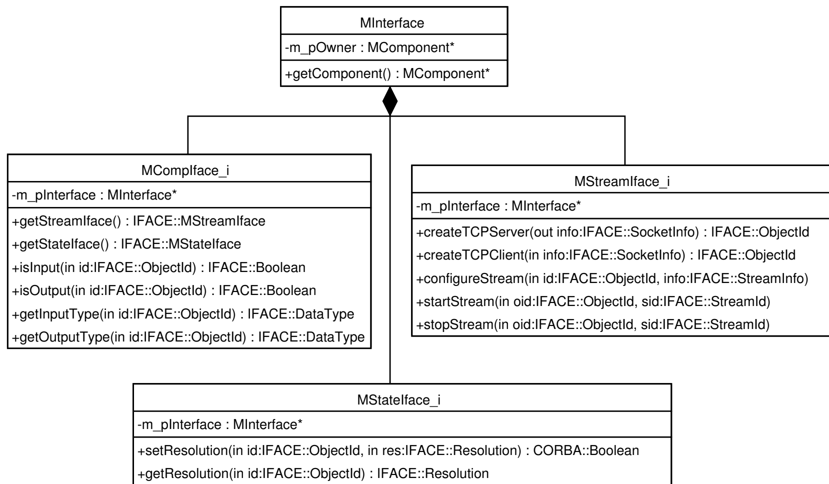

Figure 3.2: UML class diagram for CORBA interface.

Component Interface

The component interface `MCompIface` provides one single CORBA object reference to clients. A CORBA object reference to `MCompIface` is the implementation of the CORBA component reference described in Section 2.1.1 and is used for registration at the CORBA Naming Service. The interface defines operations for acquiring CORBA object references to the other interfaces. In addition, the interface provides operations to query individual objects contained by the component and to query the component as a whole. The interface is identical for all components. Here follows the CORBA IDL code[3]:

```
1.   module IFACE
2.   {
3.      interface MCompIface
4.      {
```

[3] Detailed definitions of data types and exceptions are omitted for brevity.

```
5.          MStreamIface
6.          getStreamIface();
7.
8.          MStateIface
9.          getStateIface();
10.
11.         ObjectIds
12.         getObjects();
13.
14.         boolean
15.         isInput(in ObjectId nTarget)
16.         raises(InvalidObjectId);
17.
18.         boolean
19.         isOutput(in ObjectId nTarget)
20.         raises(InvalidObjectId);
21.
22.         DataType
23.         getInputType(in ObjectId nTarget)
24.         raises(InvalidObjectId);
25.
26.         DataType
27.         getOutputType(in ObjectId nTarget)
28.         raises(InvalidObjectId);
29.       };
30.    };
```

Clients obtain CORBA object references to the stream and state interface by invoking `getStreamIface` and `getStateIface` respectively (line 5 to 9). Clients query for the multimedia objects contained by the component through the `getObjectIds` operation that returns a sequence of object identifiers (line 11 to 12).

The remaining four operations query individual objects contained by the component. The object identifier of the target object is passed as an argument. The operations `isInput` and `isOutput` return a boolean value that determines the object's type (line 14 to 20). The operations `getInputType` and `getOutputType` return the input and output data type of the target object (line 22 to 28). All four operations throw an `InvalidObjectId` if the component does not contain an object with the identifier passed as an argument.

Stream Interface

The stream interface `MStreamIface` provides operations for configuring streams between objects. The stream interface is identical for all components. Here follows the CORBA IDL code:

```
1.   module IFACE
2.   {
3.      interface MStreamIface
4.      {
5.        ObjectId
6.        createTCPServer(out SocketInfo info)
7.        raises(SocketException);
8.
9.        ObjectId
10.       createTCPClient(in SocketInfo info)
11.       raises(SocketException);
12.
13.       void
14.       configureStream(in ObjectId nTarget, in StreamInfo info)
15.       raises(InvalidObjectId, InvalidStreamInfo);
16.
17.       void
18.       startStream(in ObjectId nSource, in StreamId nStreamId)
19.       raises(InvalidObjectId);
20.
```

```
21.       void
22.       stopStream(in ObjectId nSource, in StreamId nStreamId)
23.       raises(InvalidObjectId);
24.    };
25. };
```

Clients configure a distributed stream by creating specialized TCP objects which are automatically connected. The operations `createTCPServer` and `createTCPClient` create such objects (line 5 to 11). The `SocketInfo` structure holds connection specific information, i.e. IP host address and port number. The framework treats TCP objects as regular multimedia objects and assigns them an object identifier upon creation. The object identifier is returned to the client for later use. Distributed streams are covered in Section 3.4.4.

Clients prepare a multimedia object for streaming by invoking the operation `configureStream` (line 13 to 15). The `StreamInfo` structure holds stream specific information such as stream identifier, destination object and stream priority. Clients start and stop a stream by invoking `startStream` and `stopStream` respectively. These operations require an object identifier referring to a *source* object and the stream identifier of the stream passed as arguments (line 17 to 23). Data streams are covered in detail in Section 3.4.

State Interface

The state interface `MStateIface` provides operations for accessing object specific characteristics. The state interface varies from component to component, depending on the multimedia objects it contains. Component developers are responsible for the design and implementation of the interface. Here follows an example IDL:

```
1.   module IFACE
2.   {
3.      struct Resolution {
4.        unsigned short nHorizonal;
5.        unsigned short nVertical;
6.      };
7.
8.      interface MStateIface
9.      {
10.       Resolution
11.       getResolution(in ObjectId nTarget)
12.       raises(InvalidObjectId);
13.
14.       bool
15.       setResolution(in ObjectId nTarget, in Resolution res)
16.       raises(InvalidObjectId);
17.    };
18. };
```

This interface applies to multimedia objects for accessing its resolution, for example a camera or display object. The object identifier passed as an argument to the operations `getResolution` and `setResolution` must refer to an object supporting the interface. If not, an `InvalidObjectId` exception is thrown.

3.2.2 Multimedia Objects

The multimedia framework provides direct support for sources, filters and sinks by means of the classes `MSource`, `MFilter` and `MSink` respectively. An object developer extends these classes and implements the provided (pure) virtual methods to add custom behavior. Once an object is developed it can be reused in any other component. Hence, new components

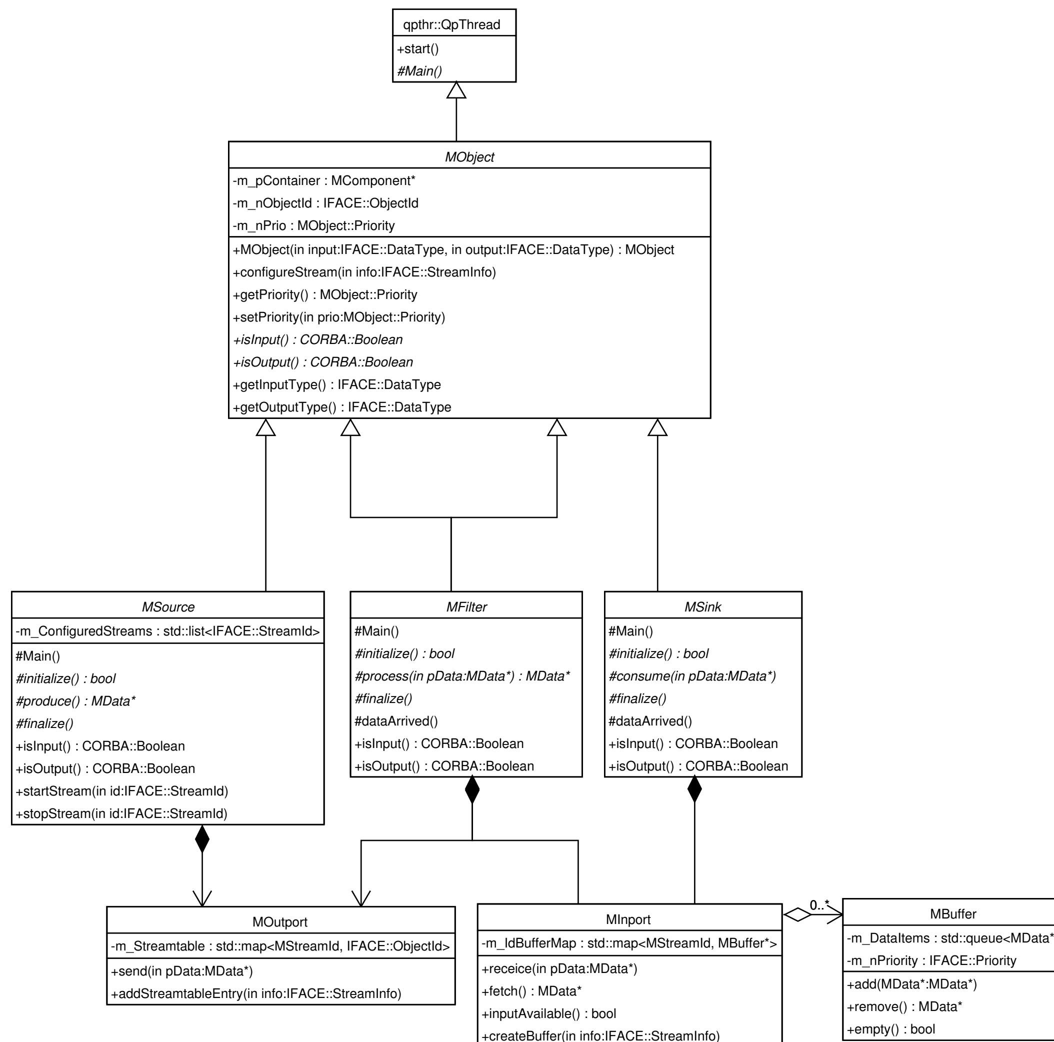

Figure 3.3: UML class diagram for multimedia objects.

can be composed from existing objects with little effort. Figure 3.3 depicts inheritance and composition relations among the classes that contribute to `MSource`, `MFilter` and `MSink`. In the remainder of this section all classes will be explained in a top-down manner.

Base Class

All multimedia objects share the same base class `MObject`. This class derives from `QpThread` that is part of the object-orientated QpThread library, a C++ wrapper for Posix threads, providing basic thread functionality. The class `QpThread` declares the protected virtual method `Main` that is executed in a separate thread after `start` is invoked on a multimedia object. Object behavior varies for sources, filters and sinks. Therefore, the `Main` method is implemented by the corresponding class.

Multimedia objects have an input and output data type. When clients configure a stream, the input and output types of two consecutive objects in the stream must match. The methods

`getInputType` and `getOutputType` return the object's input and output type initialized in the object's constructor.

Before starting a stream, a client must prepare all objects involved in that stream. A client can prepare an object for a data steam by invoking the method `configureStream`. The method requires one argument that is a structure holding stream specific information such as stream identifier, destination object and stream priority. Data streams are covered in detail in Section 3.4.

Priority values can be assigned to objects as a criteria for preemption since multiple objects may compete for the CPU simultaneously. By assigning priority values, the underlying operating system decides which thread utilizes most CPU cycles during execution. The methods `getPriority` and `setPriority` get and set the priority value of an object respectively.

Inports and Outports

As described in Section 2.1.2, multimedia objects are categorized as input and/or output objects. An output object has an outport for sending multimedia data to input objects and an input object has an inport for receiving data from output objects. Naturally, matching input and output types are required.

Outports are implemented by the `MOutport` class. An outport manages a stream table that holds tuples of type `[MStreamId, IFACE::ObjectId]` meaning that data belonging to the stream with id `MStreamId` is sent to the local object with id `ObjectId`. Stream table entries are added through the `addStreamtableEntry` method. Its only parameter is a structure holding stream specific information such as stream identifier, destination object and stream priority. There are no methods for deleting entries from the stream table; the framework deletes obsolete entries automatically after a certain period of inactivity. Clients need only to send a `stopStream` message to the source to stop a data stream instead of notifying all involved objects. As a result, data items for the respective stream are no longer produced and stream table entries will not be referenced and finally removed. This reduces the number of CORBA messages sent over the network.

Inports are implemented by the `MInport` class. An inport manages one separate buffer for each configured stream. Each stream identifier has an associated buffer stored in the STL-map data structure that holds tuples of type `[MStreamId, MBuffer*]`. When a data item is received, it is added to the buffer identified by the `MStreamId` variable. Buffers are created through the `createBuffer` method. Its only parameter is a structure holding stream specific information such as stream identifier, destination object and stream priority. There are no methods for deleting buffers from inports. Buffers are automatically deleted after a certain period of inactivity rather than explicitly deleted by the client. Motivation for automatic cleanup is identical to the motivation for the removal of entries from stream tables.

The `MBuffer` class is more than just a wrapped STL container class. The class implements an algorithm for the discard of obsolete data items. In short, the algorithm decides whether the next data item in the buffer is obsolete by comparing the elapsed time between the last and current fetch with the difference in timestamps between the last fetched item and the next item in the buffer. If the former is larger than the latter, the data item is discarded and the next item is checked until the condition is satisfied or the buffer is empty.

An alternative to the one-to-one relationship between input buffers and data streams would be one single input buffer for all configured data streams. Two problems with this approach were found. First, the recording of the time difference between two consecutive data items

belonging to one stream becomes unfeasible. Second, for the implementation of stream priority, the framework needs to know which streams are configured for the object when a data item is fetched from the inport.

Sources, Filters and Sinks

Multimedia object developers derive from `MSource`, `MFilter` and `MSink` to implement sources, filters and sinks respectively. The classes provide virtual methods that are to be implemented in derivations in order to add custom behavior. The framework implements a different `Main` method for `MSource`, `MFilter` and `MSink`, that is executed in a separate thread. Inside `Main`, the virtual methods are invoked internally. Hence, execution is temporarily transferred outside the framework to the developer's code.

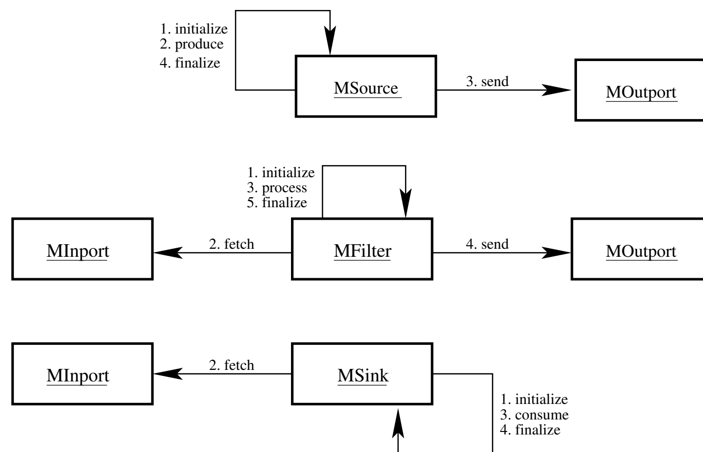

Figure 3.4: Object execution for sources, filters and sinks.

Object execution, realized by invoking `start` on a multimedia object and resulting in a call to `Main`, is represented by the UML collaboration diagrams depicted in Figure 3.4[4]. For example, the `MFilter::Main` method executes the following sequence until object execution is explicitly stopped:

1. **Initializing.** The virtual method `initialize` is invoked and execution is temporarily transferred outside the framework to developer's code for object initialization.
2. **Data Fetching.**
   (a) If object execution is explicitly stopped, goto 5.
   (b) If data is available, a data item is fetched from one of its input buffers by invoking `fetch` on its inport. Otherwise, the execution thread is put to sleep until it is notified upon data arrival.
3. **Data Processing.** The pure virtual method `process` is invoked and execution is transferred outside the framework. Developer's code processes the data passed as input parameter and returns the result.

[4]Source code for object execution with explanation can be found in Appendix B.

4. **Data Sending.** The filter calls `send` on its outport passing the processed data as an argument. The outport is responsible for sending the data to its target object. Goto 2.

5. **Finalizing.** The virtual method `finalize` is invoked and execution is temporarily transferred outside the framework to developer's code for object finalization.

Stream identifiers are used as entries in stream tables managed in outports. When data arrives in an outport, the identifier that is sent as part of the data is extracted and a lookup in the stream table produces the identifier of the target object. In addition, stream identifiers are used in inports for identifying buffers. When data arrives in an inport, the identifier is extracted from the data and used as a buffer identifier. Motivation for stream identifiers is explained in Section 3.4. Sources manage a list of configured stream identifiers and produce data items for each stream. Consequently, adding and removing a stream identifier from the list is semantically equivalent to starting and stopping a data stream. The methods `startStream` and `stopStream` are designed for this purpose and require a stream identifier as an input parameter.

3.2.3 Remote Method Invocation

The alert reader might have noticed that some operations in the CORBA interface have an identical name to a method declared in one of the classes `MObject`, `MSource`, `MFilter` or `MSink`. Such CORBA operations require an object identifier as an argument that is used to acquire a C++ pointer from the container component. If such an object exists, the method is invoked and the result is returned by the CORBA operation. Otherwise, an `InvalidObjectId` exception is thrown.

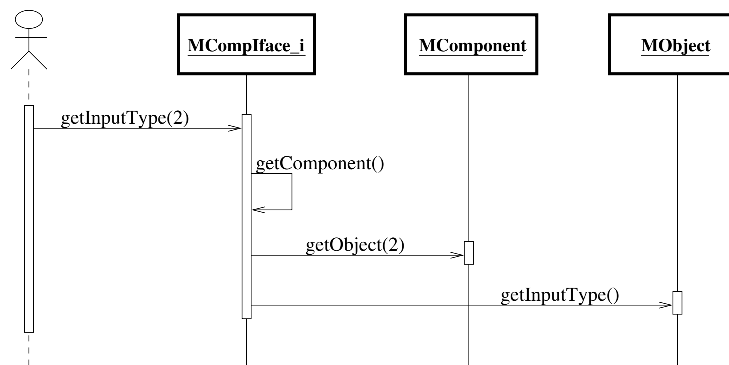

Figure 3.5: UML sequence diagram for remote method invocation.

For example, consider the method invocation `getInputType` on the CORBA component interface in Figure 3.5. The object identifier `2` is passed as an argument to the method. The interface obtains a reference to its component and calls `getObject` on this reference, providing the object identifier as an argument. As a result, the C++ pointer referring to the multimedia object is returned. The pointer to the object is used to get the input type of the object and the result is finally returned to the client.

3.3 Object Interaction

In this section the interaction of multimedia objects contained by one component will be described. Since all objects run in separate threads and different objects may access the same data structures simultaneously, concurrency need to be accounted for. Moreover, the threads have to be synchronized in order to avoid busy waiting and overflowing buffers. To explain how multimedia objects address these issues, the example component depicted in Figure 2.2 is explored.

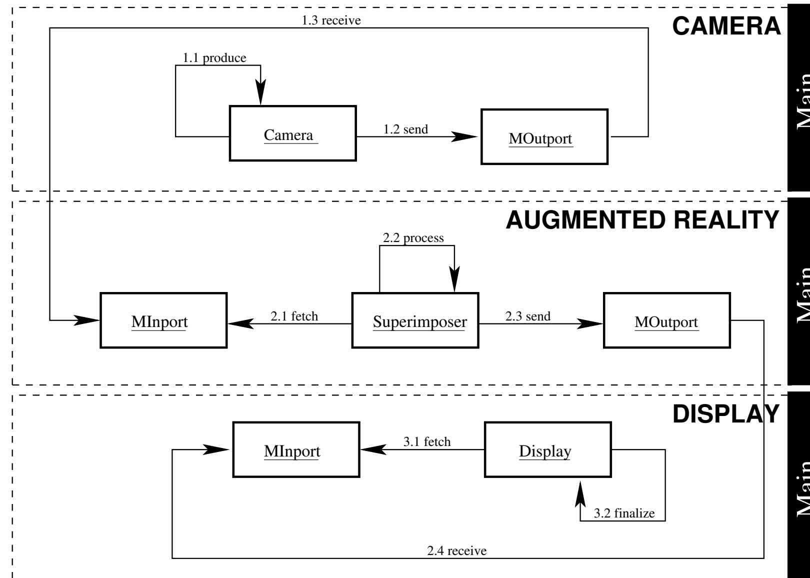

Figure 3.6: UML collaboration diagram for object interaction.

In the example, data is streamed from a camera to a display object through an augmented reality object. The camera object is a source that produces data by capturing images from a video device. The augmented reality object is a filter that processes data by super imposing digital images. The display object is a sink that consumes the data by showing the video images onto the screen. All objects execute their own `Main` method in a separate thread. Figure 3.6 shows the execution sequence of the three main methods (the `initialize` and `finalize` methods are omitted for clarity) and how they communicate.

3.3.1 Communication

The sequences 1, 2 and 3 are executed in parallel. Sequence 1 produces video frames (1.1) and sends them to the outport belonging to the camera object (1.2). In turn, the outport calls receive on the inport owned by the augmented reality object (1.3). Consequently, video frames are added to the input buffer for that stream managed in the inport. At the same time the augmented reality object might access the same input buffer by calling `fetch` on its inport (2.1). To avoid race conditions, the input buffer guarantees mutual exclusion. The operations `receive` (2.4) and `fetch` (3.1) synchronize access to the input buffer belonging to the camera object in the same way.

3.3.2 Synchronization

Suppose the augmented reality object processes video images at a faster rate then it receives images in its input buffer. Then, after processing one image, it will find an empty buffer when fetching the next data item. To avoid busy waiting, the thread is put to sleep until it is notified by the camera thread when it receives data in its inport (1.3). In this way, multimedia data functions as a scheduling token for multimedia objects.

On the other hand, if the augmented reality object receives data faster than it can process, its buffers will overflow and the container component eventually will crash. It was explained in Section 2.3 that the discarding of obsolete data items solves the problem of overflowing buffers. When `fetch` is called on an inport belonging to an input object (2.1/3.1), obsolete data items are deleted from input buffers before the next data item is returned.

3.4 Data Streams

Filters can be configured for multiple streams at the same time (see Section 2.2.5). As a result, data items belonging to different streams might have different destinations. So how does a filter know where to send each data item? Clearly, setting up direct connections between multimedia objects does not resolve the issue.

One solution would be to add a routing list containing universal object identifiers to each data item, defining the streaming route. A source would add such a list to each data item it produces and send it as part of the data structure. When a data item arrives in the outport belonging to an output object, the head is removed from the routing list and the data is sent to the corresponding object. Hence, the routing list would be empty when it arrives in the sink. However, three problems were found with this particular solution:

1. An intermediate filter might want to broadcast its outflowing data to more than one target object. This implies a simple routing list would not be sufficient and a more complex data structure such as a routing tree is needed to solve the issue.

2. Adding a routing tree consisting of universal object identifiers to each produced data item results in a much larger data structure sent over the network.

3. Clients may want to assign priorities to competing streams. This implies that data needs to be identified to be part of one specific stream. Routing trees do not provide help for the identification of streams.

To circumvent these three problems, the framework uses unique *stream identifiers* instead of routing lists. If a source is configured for a stream, the framework adds the stream identifier to the multimedia data structure after it is produced. When the data item arrives in the outport belonging to an output object, the stream identifier is extracted from the data. The outport uses the identifier for a lookup in its stream table and finds the target object identifier for sending the data.

This solution is extended for cyclic and broadcasting streams. In the remainder of this section, stream identifiers with respect to data streams will be clarified by illustrative examples of each type of stream supported by the framework (see Section 2.2).

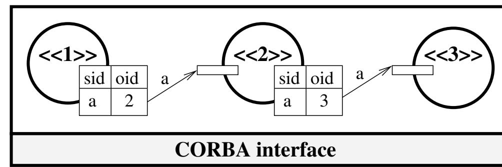

Figure 3.7: Stream tables for a basic stream.

3.4.1 Basic Streams

The solution outlined above does not need any extension for the basic stream type. By way of illustration, consider the example basic stream in Figure 3.7. The source `<<1>>` adds the stream identifier `a` to each produced data item. When arriving in its outport, the stream identifier is extracted from the data and a lookup in its stream table produces the target object identifier `<<2>>`. The outport retrieves a pointer to the actual object from its container component providing the object identifier, and calls `receive` on the inport belonging to that object. The filter `<<2>>` is notified that data has arrived in its input buffer and retrieves the data item for processing. After processing, the data item is handled by its outport in the same way as for the source.

3.4.2 Cyclic Streams

For the cyclic stream type the solution for data streaming needs an extension. Suppose that a filter is visited by the same stream twice. Then the stream table would contain two entries for the same stream identifier. But how would the filter know which entry to use when data is to be sent by its outport? To solve the issue, redefinition of a stream identifier as a tuple of type `[IFACE::StreamId, IFACE::SubStreamId]`[5] is necessary. The sub-stream identifier is initialized to zero in the source. Now, before data is sent by an outport belonging to an output object, the sub-stream identifier is incremented by one. In this way, filters can distinguish data items that visit the object for the second time.

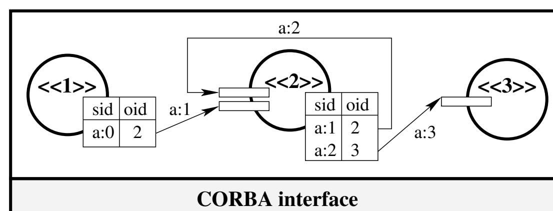

Figure 3.8: Stream tables for a cyclic stream.

For illustration, consider the example cyclic stream in Figure 3.8. The source `<<1>>` initializes the sub-stream identifier zero. When data arrives in its outport, a lookup in the stream table with identifier `[a,0]` produces the object identifier `<<2>>`. Before the data is sent to the inport belonging to the target object, the sub-stream identifier is incremented. Hence, after data is processed by the filter `<<2>>` for the first time, a lookup in its stream table with value `[a,1]` will produce object identifier `<<2>>` again, referring to itself. Before sending the

[5]Such tuples plus utility methods are implemented by the MStreamId class used in stream tables and as buffer identifiers.

data to its own inport, the sub-stream identifier is incremented again. Consequently, when data arrives in the inport belonging to the filter for the second time, the identifier will equal [a,2] and a lookup will produce object identifier <<3>>. Clearly, this breaks the cycle and data is finally sent to the sink, after incrementing the sub-stream identifier one more time.

3.4.3 Broadcasting Streams

The solutions for data streaming needs another extension for the broadcasting stream type. A naive, but correct approach is to treat each branch in a broadcasting stream as a basic stream. A source object produces one data item for each stream and adds a unique stream identifier to the data structure. The problem with this approach is that copies of the same data will be identically processed by filters (see Section 2.2.3).

A more efficient approach is to bundle basic streams into one before it is broadcasted. After the broadcast, the streams are unbundled and go their own way. Therefore, a *list* of stream identifiers is added to the data structure after it is produced by the source. Each element in the list would be the identifier of one bundled stream. When data arrives in the outport of an output object, the list is extracted from the data and partitioned into parts for different destination objects. Consequently, each partition resembles one branch in the broadcasting stream. For each partition the data is cloned and sent to the corresponding destination object.

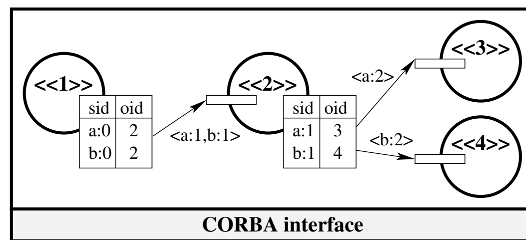

Figure 3.9: Stream tables for a broadcasting stream.

For the purpose of illustration, consider the example broadcasting stream in Figure 3.8. After data is produced by source <<1>>, the framework adds the list of configured stream identifiers, [a,0] and [b,0], to the data structure. In the outport of the source, the partitioning of the list results in one partition since both stream identifiers share the same destination object. The streams are bundled and just one copy of the data is sent to filter <<2>> for processing. When the data arrives in the outport of the filter, the partitioning of the list results in the two partitions, because both stream identifiers in the list have different destination objects. Consequently, data is cloned and each item is assigned one partition before it is sent.

3.4.4 Distributed Streams

The solution for data streaming does not need an extension for distributed streams. However, there are other design issues that have to be addressed when transmitting data between components. When using the Internet Protocol (IP), there is a choice between two levels of service: *streams*[6] and *datagrams*. Next will be a discussion which service best suites the

[6]Note that streams as a level of service and data streams used in the multimedia framework are two different concepts.

purposes discussed here and how the service is incorporated into the framework.

Streams vs Datagrams

A *stream socket* establishes and maintains two-way byte stream connections and guarantees that data is not lost, duplicated or reordered at the price of slow startup and resource consumption. Stream sockets are implemented by TCP/IP connections. In contrast, a *datagram socket* does not establish and maintain connections but provides an unsequenced and unreliable service. That is, data packages may get lost, duplicated or arrive out-of-order. On the other hand, the service is relatively inexpensive in terms of resources and also fast because no connections are established and maintained. Datagrams are implemented by UDP connections.

The stream service automatically partitions large data segments on the sender side and little effort is required to reassemble the data segment on the receiver side because the service guarantees in-order data arrival without loss and duplication. Datagrams have a maximum size much smaller than the data items sent between components. On the sender side, data must be divided into packages that fit into a datagram by the programmer. On the receiver side, the data has to be reassembled accounting for package loss, duplication and out-of-order arrival. Luckily, RTP[8] libraries implemented on top of UDP have been developed to alleviate these matters. Still, relatively speaking, much more effort is required from the programmer.

Components typically run on platforms with varying performance. In case of the stream service, the slowest participant in the TCP/IP connection sets the pace of data transmission automatically. In case of the datagram service, the sender and receiver have to synchronize the transmission speed of datagrams because of its connectionless nature. Low-performance receivers may not be able to process datagrams as fast as they are sent by high-performance senders. The RTP library provides sender and receiver reports that are automatically exchanged between participants. The reports include all necessary information, but programmers have to account for complex algorithms for adjusting the transmission speed to reflect network behavior.

Components may suddenly become unavailable due to system or network failure. In the case of the stream service, both participating components need to be restarted to recover from the failure. In the case of the datagram service the non-failing component can continue executing since no connections are broken.

After experimenting with RTP it was decided to switch to TCP/IP mainly because of its simple usage. The framework is still in its early stages and it was found that complex issues such as synchronizing senders and receivers can be incorporated in the framework in a later stage. Also, it was found that TCP/IP is sufficient in terms of efficiency for data items roughly not exceeding 200 KB on a 100 Mbps network, enough for transmitting medium-sized video images.

TCP Servers and TCP Clients

For the implementation of inter-component data streaming the framework provides two types of specialized multimedia objects: *TCP servers* and *TCP clients*. One TCP client is connected to one TCP server by a two-way byte stream. A TCP client encodes multimedia data into a byte stream and sends it to its connected TCP server that decodes the byte stream into the data object (of course `decode(encode(d)) = d` for all data items `d`).

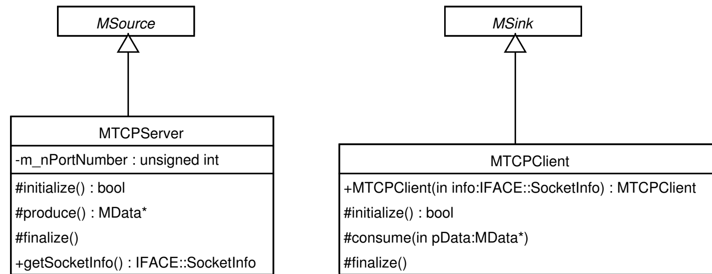

Figure 3.10: UML class diagram for TCP servers and clients.

Consider the UML class diagrams in Figure 3.10. `MTCPServer` derives from `MSource` and `MTCPClient` from `MSink`. Consequently, TCP servers and clients are classified as output and input object respectively. Semantically it is not strange at all to derive `MTCPServer` from `MSource`, since from the component's point of view data is produced by receiving it from another component. Analog reasoning applies to the `MTCPClient` class.

In its constructor `MTCPServer` creates a socket for a specific port number. After object execution, the `initialize` method is executed first and halts, waiting for an incoming connection. The port number and the IP host address of the server are passed to the `MTCPClient` constructor. After object execution, its `initialize` method establishes a connection using the provided socket information. From there on, data is streamed over TCP/IP between the objects. Since a TCP object derives from `MSource` or `MSink`, the framework treats them as regular multimedia objects. Consequently, TCP objects can be configured for local streams.

Example Distributed Stream

By way of illustration, consider the example distributed stream in Figure 3.11. The TCP client `<<2>>` in the left component is configured for one inflowing data stream and receives data from the source `<<1>>`. In its `consume` method, the object encodes data items into a byte stream and sends the byte stream to the TCP server `<<1>>` in the right component over TCP/IP. In its `produce` method, the TCP server receives the byte stream and decodes it into a multimedia data object. Afterwards, it sends the data to the sink `<<2>>` that consumes the stream.

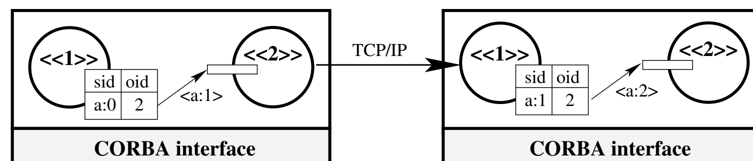

Figure 3.11: Stream tables for a distributed stream.

3.4.5 Competing Streams

In this subsection it will be explained how stream identifiers contribute to the implementation of stream priorities for competing streams. As said before, every input object manages one input buffer for each configured stream in its inport. In addition, each input buffer is assigned

a priority value upon creation that is equal to the priority value of the stream it belongs to. When an input object fetches a data item from its inport, it applies a selection algorithm for choosing the input buffer based on the priority values. In this way, buffers with higher priorities are more likely to be chosen for delivering the next data item. If not designed carefully, the selection algorithm might cause *buffer starvation* - buffers with low priority might never be selected - resulting in overflowing buffers.

In case of broadcasting streams where several streams may be bundled into one, one buffer for each stream in the bundle is created rather than one buffer for the bundled stream as a whole. When data arrives in the inport belonging to an input object, the data is placed in the input buffer with highest priority among the bundled streams. The reason for this approach is that streams are dynamical by nature. Hence, changes in broadcasting streams are reflected without additional effort. Suppose one of the streams in a bundled stream is stopped, then buffers corresponding to the remaining streams are still available and can be used immediately. If one buffer for the bundled stream as a whole is used, changing a broadcasting stream would imply creating a new buffer.

3.5 Multimedia Data

When designing a multimedia data type several issues need to be addressed. These issues described in Sections 2.3 and 3.4 lead to the following requirements:

- encoding into a byte stream and decoding from a byte stream for sending data over the network,
- timestamping for the discard of obsolete data in input buffers,
- cloning for broadcasting streams,
- managing a list of stream identifier for the implementation of data streams,
- generality and extensibility for custom data types.

Putting these requirements together the base class for multimedia data in Figure 3.12 was decided on. The framework deals with pointers to the base class exclusively. Developers can design custom data types by extending the base class adding custom data members and overriding the necessary abstract methods.

Functionality for time stamping and stream identifiers is completely handled by the base class and does not require any effort from developers of new data types. In contrast, encoding and decoding of custom data types requires some additional effort. The base class handles the serialization of the base members but has no knowledge of custom data members added in derivations. To assure correct serialization a developer requires to override both methods `encode` and `decode` and call the base class method in its first statement before serializing its own data members.

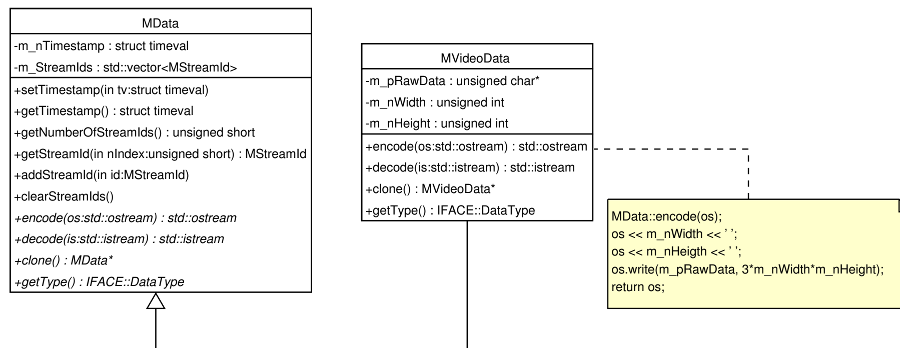

Figure 3.12: UML class diagram for multimedia data.

In addition, cloning of multimedia data cannot be handled by the base class by any means. Remember the framework deals with pointers to `MData` exclusively. When data is multiplied in response to a broadcast, the framework has no knowledge about the type of data and consequently cannot decide which copy constructor to call for. A variation of the Prototype Design Pattern[1] circumvents a direct call to the constructor by invoking the pure virtual method `clone` on the data object. Derivations of the method simply return a copy of itself by calling its own copy constructor.

Chapter 4

Usage

The multimedia framework, described in the previous chapters, is designed for easy usage and fast development of multimedia components and objects. To clarify how this aim is realized this chapter describes a step-by-step method for building components by using the example depicted in Figure 2.2. The steps involved are object implementation, component composition, component execution and stream configuration. This method can be used as a general guideline for building more complex configurations.

4.1 Object Implementation

Multimedia objects are implemented by deriving from `MSource`, `MFilter` or `MSink` and overriding the corresponding virtual methods for implementing custom behavior. Source code for the augmented reality object in Figure 2.2 might look something like this:

```
1.   #include "mmf.hh"
2.
3.   class AugReal :
4.      public MFilter
5.   {
6.      public:
7.
8.         AugReal(unsigned int nXPos, unsigned int nYPos, const string& strImage);
9
10.      protected:
11.
12.         virtual bool initialize();
13.         virtual MData* process(MData* pData);
14.
15.     private:
16.
17.        unsigned int m_nXPos, m_nYPos;
18.        string m_strImage;
19.        MRGBImage m_Image;
20.  };
21.
22.  AugReal::
23.  AugReal(unsigned int nXPos, unsigned int nYPos, const string& strImage):
24.     MFilter(VIDEO_DATA, VIDEO_DATA),
25.     m_nXPos(nXPos),
26.     m_nYPos(nYPos),
27.     m_strImage(strImage)
28.  {  }
29.
30.  bool
```

```
31. AugReal::
32. initialize()
33. {
34.    bool bSuccess = true;
35.    // read m_strImage from database
36.    return bSuccess;
37. }
38.
39. MData*
40. AugReal::
41. process(MData* pData)
42. {
43.    MVideoData* pVideoData = dynamic_cast<MVideoData*>(pData);
44.    // superimpose m_Image onto video frame
45.    return pVideoData;
46. }
```

In the constructor the input and output type are passed to the `MFilter` base class. In addition, the position for superimposing the digital image and the name of the image are initialized (line 22 to 28).

The class overrides the two virtual methods `initialize` and `process` from the `MFilter` base class. In the `initialize` method, the image that is superimposed onto video frames is read from a database. The method returns a boolean value to the framework denoting whether it was completed successfully or not (line 30 to 37).

In the `process` method, first the input parameter is downcasted to the expected data type and thereafter the digital image is superimposed onto the video frame. Finally, the resulting image is returned to the framework (line 39 to 46).

4.2 Component Composition

One way to compose a component is to derive from the `MComponent` class. In its constructor, multimedia objects are created and added to the component. Source code for the example component in Figure 2.2 might look something like this:

```
1.  #include "mmf.hh"
2.  #include "camobj.hh"
3.  #include "augobj.hh"
4.  #include "disobj.hh"
5.
6.  class MyComponent :
7.     public MComponent
8.  {
9.     public:
10.
11.       MyComponent(const string& strName);
12.
13.    private:
14.
15.       Camera* m_pCamera;
16.       AugReal* m_pAugReal;
17.       Display* m_pDisplay;
18. };
19.
20. MyComponent::
21. MyComponent(const string& strName):
22.    MComponent(strName)
23. {
24.    m_pCamera = new Camera;
25.    m_pAugReal = new AugReal;
26.    m_pDisplay = new Display;
27.
28.    addObject(m_pCamera);
```

```
29.     addObject(m_pAugReal);
30.     addObject(m_pDisplay);
31.  }
```

The constructor takes a string argument that is passed to the `MComponent` base class for registration at the CORBA Naming Service. In its body, the multimedia objects are created first and added to the component thereafter (line 20 to 31).

The multimedia objects that are added to the component are developed in earlier stages of the development process. In general, objects can be reused in any component, clearly reducing development time and costs.

4.3 Component Execution

For the execution of a component a multimedia environment is initialized first. I.e. a local CORBA Object Request Broker is created, the QpThread library is initialized and the service level for debug logging is set. Afterwards, the component can be created and started. Source code for the execution of the example component in Figure 2.2 might look something like this:

```
1.   #include "mmf.hh"
2.   #include "mycomp.hh"
3.
4.   int
5.   main(int argn, char** argc)
6.   {
7.      if(argn != 2) {
8.         cerr << "Usage: main <name>" << endl;
9.         return -1;
10.     }
11
12.     try {
13.         MInit init(argn, argc);
14.         MyComponent mycomp(argc[1]);
15.         mycomp.start();
16.     }
17.     catch(const MCorbaExc& exc) {
18.        cerr << exc << endl;
19.     }
20.  }
```

A user executes the component by running `main` from the command line. The method requires the component's name as an argument and therefore correct usage is checked (line 7 to 10). If successful, the multimedia environment is initialized, the component is created on the stack and executed by invoking `start` (line 13 to 15). The creation of an Object Request Broker or component registration at the CORBA Naming Service might fail resulting in a `MCorbaExc` exception (line 17).

Generally speaking, identical components can be executed on different machines at the same time. The user has to provide a different name for each component to avoid name clashes at the CORBA Naming Service, resulting in a `MCorbaExc`. Like multimedia objects, components need to be developed only once, clearly decreasing development time and costs.

4.4 Stream Configuration

For the configuration of a stream between multimedia objects a client obtains the involved CORBA component references from the CORBA Naming Service. Object identifiers of the

multimedia objects contained by the components are acquired by invoking the appropriate method on a CORBA component reference. Together they form universal object identifiers used for the configuration of a stream by means of the `MStream` utility class. This class is an implementation of the Facade Design Pattern[1] and provides an abstraction to low-level operations defined in the CORBA stream interface. Source code for the configured stream in the example component in Figure 2.2 might look something like this:

```
1.  #include "mmf.hh"
2.
3.  int
4.  main(int argn, char** argc)
5.  {
6.     try {
7.        MORB::init(argn, argv);
8.
9.        IFACE::MCompIface_var pCompIface = MNaming::resolve<IFACE::MCompIface>("some name");
10.       if(CORBA::is_nil(pCompIface)) {
11.          cerr << "Component not registered at CORBA Naming Service" << endl;
12.          return -1;
13.       }
14.
15.       MStream stream(MRandom::generate(0, UINT_MAX), MStream::NORMAL);
16.       stream.setSource(pCompIface, 1);
17.       stream.addFilter(pCompIface, 2);
18.       stream.setSink(pCompIface, 3);
19.       assert(stream.check());
20.       stream.start();
21.    }
22.    catch(const MCorbaExc& exc) {
23.       cerr << exc << endl;
24.       return -1;
25.    }
26.    catch(const MStreamExc& exc) {
27.       cerr << exc << endl;
28.       return -1;
29.    }
30.
31.    return 0;
32. }
```

For accessing the CORBA Naming Service a local Object Request Broker is initialized (line 7). The CORBA component reference is obtained by invoking the static method `resolve` on the `MNaming` utility class and the result is checked for failure; if no component is registered under the provided name, the method returns a nil-reference (line 9 to 13).

An `MStream` object is created by passing a unique stream identifier and a priority value as arguments to its constructor (line 15). The stream is configured by invoking the appropriate methods on the `MStream` object passing the universal object identifiers[1] as parameters (line 16 to 18). The stream configuration is checked for the validity of the universal object identifiers, i.e. for correct type of objects and matching input and output types (line 19). Finally, the stream is started (line 20).

Two types of exceptions can be thrown by the framework: `MCorbaExc` and `MStreamExc` (line 22 to 29). The former indicates initialization failure of the CORBA Object Request Broker, or communication failure with the CORBA Naming Service or individual components. The latter indicates a malformed stream and should not be thrown if the assertion in line 19 is satisfied.

[1] Note that in this example the object identifiers are known a priori.

Chapter 5

Framework in Operation

Reusing multimedia objects simplifies and accelerates the development of new components. Many augmented reality features, such as detecting visual markers and superimposing digital images, are common in most applications. The DCL research department developed an *augmented reality class library* that implements such common services. The class library is introduced briefly in Section 5.1.

Augmented reality applications are developed relatively easy by reusing multimedia objects from the class library. Section 5.2 shows how the library is used. This is illustrated by describing two example applications that are built using the DCL middleware, one of them reusing objects from the library.

5.1 Augmented Reality Class Library

The augmented reality class library is developed on top of the multimedia framework as depicted in Figure 5.1. It contains 12 classes with a total of 58 methods, excluding (copy-) constructors. The library provides support for extended data types, visual marker detectors, location sensors and image renderers which will be explained briefly in the next sections.

5.1.1 Extended Data Type

Augmented reality applications use marker information intensively. The `MRVideoData` class is a specialization of `MVideoData`, provided by the multimedia framework, which adds marker information to its base class. To address the differences among marker information used by third-party libraries the class holds a pointer to the abstract base class `MRMarkerInfo` that defines a general interface. An `MRVideoData` object is configured with a concrete derivation of the abstract class (Strategy Design Pattern[1]). Currently, the augmented reality library supports two concrete types of marker information, one based on TRIP and one based the ARToolkit.

5.1.2 Detectors

For the detection of visual markers in video frames the augmented reality library defines the `MRDetector` class that is a derivation from `MFilter`, provided by the multimedia framework.

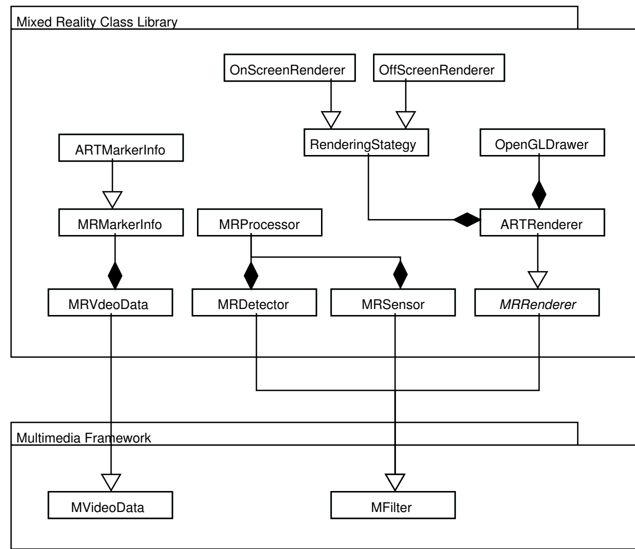

Figure 5.1: Augmented reality class library.

Third-party libraries differ in marker detecting algorithms. To address such differences an `MRDetector` object is configured with a concrete derivation from `MRProcessor` which defines a general interface (Strategy Design Pattern[1]). Currently, concrete processors based on TRIP (ARToolkit) expect video data configured with TRIP (ARToolkit) marker information. These restrictions are the result of the marker specific characteristics of the detection algorithms. At present, adapting the detection algorithm for more general marker information is studied and might be incorporated in the near future.

5.1.3 Sensors

Sensors measure the physical location of visual markers and therefore are very similar to detectors. The location of an object with an attached marker is conveyed by the sensors and sent to the context database. The communication infrastructure uses the information for automatic reconfiguration of the application in response to context change.

5.1.4 Renderers

For superimposing digital images onto video frames the library provides one concrete renderer class `ARTkRenderer` based on the ARToolkit. The class only renders video images configured with marker information based on the ARToolkit. Therefore, marker information is converted when needed. The reason for this ad-hoc approach is that the TRIP library does not support rendering functionality. Still TRIP detectors are used because marker location sensors are much more accurate.

An `ARTkRenderer` object is configured with a concrete rendering strategy (Strategy Design Pattern[1]). Currently, the library provides support for on screen and off screen rendering and uses OpenGL for superimposing 3D images onto video frames.

5.2 Applications

Figure 5.2 illustrates how augmented reality objects from the library are configured and connected in an application. In the example, a visual marker is attached to a person. The camera object captures video frames and sends the data contained in a `MRVideoData` object to the detector. The detector object spots the marker and adds the information to the video data structure before sending it to the renderer object. The renderer object superimposes information about the person onto the video frame and sends the result to the display. The display object presents the video image on the screen.

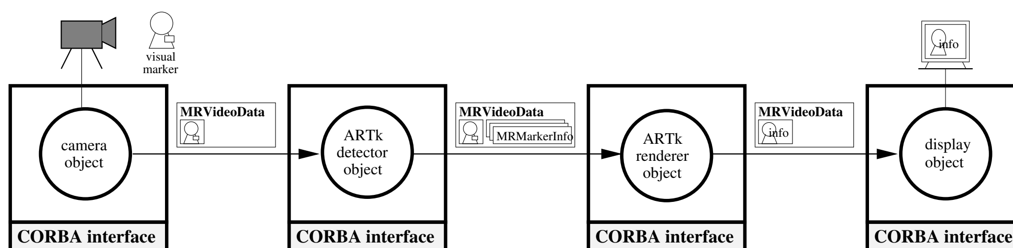

Figure 5.2: Augmented reality application.

The multimedia objects are distributed over different components running on remote machines. The detection of visual markers and the rendering of images are heavy calculations. For that reason the respective components are likely to run on high-performance machines. In another configuration, the detector and rendering object might be contained by one component removing communication overhead between the components. Next, two applications that are built using the DCL middleware will be described.

5.2.1 Follow-Me Application

The first application is the follow-me application[1]. A room is equipped with a camera connected to a computer running a camera component and several displays connected to computers running display components. As a user moves through the room, his or her location is sensed by an infrared sensor. According to the user's whereabouts, the nearest display is chosen for showing the captured images.

Detailed descriptions about the application is beyond the scope of this paper. In short though, the middleware controls the application and is notified of location change. It has access to the context database that stores the physical location of the displays in the room. If the distance to the current display becomes larger than the distance to another display in the room, the stream is reconfigured to use the nearest display.

The application uses proxy objects instead of direct references, i.e. universal object identifiers, to display objects. Such proxy objects hold a reference to the actual object and is updated when needed. Hence, the application uses the nearest display transparently.

[1] This application is typical for ubiquitous computing environments. It does not use any augmented reality features.

5.2.2 Mobile Augmented Reality

The second application is a mobile augmented reality application. Consider Figure 5.3. The Compaq iPAQ H3800 PDA in the picture is equipped with a wireless LAN and an RFID tag attached to it. The TOSHIBA IT refrigerator named *Feminity* contains sensors that let us know how many bottles are inside. When the user comes near the refrigerator, the RFID reader recognizes the RFID tag attached to the PDA. The RFID reader sends the location to the context database.

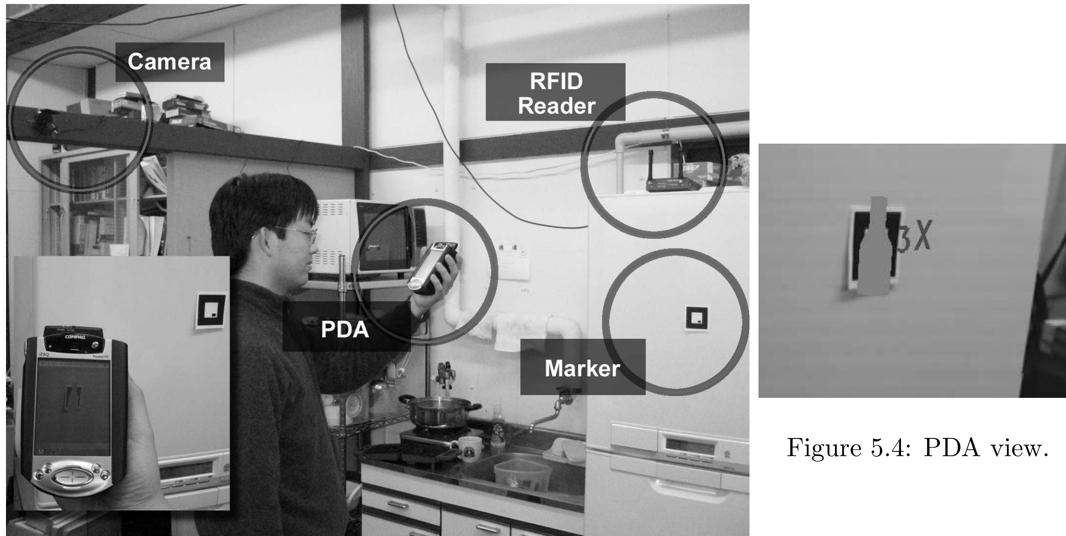

Figure 5.3: Mobile augmented reality application.

Figure 5.4: PDA view.

In addition, the refrigerator has an attached tag that is superimposed by a digital image representing the number of bottles in the refrigerator. After the RFID reader sends the PDA's location to the context database, the communication infrastructure is notified and a stream is configured to show the superimposed images on the PDA. Augmented reality components for detecting visual markers and superimposing digital images run on a nearby, powerful machine. Video frames captured by the camera are streamed to the PDA through the augmented reality objects. The result is shown in Figure 5.4.

Chapter 6

Evaluation

Development work needs serious testing. For the evaluation of this framework three test cases were examined. The first covers metrics for the discard of obsolete data, the second measures the priority algorithm applied on two competing streams and the third evaluates a distributed augmented reality application. This chapter reports the results.

6.1 Data Discard

Input buffers decide whether data items are obsolete and therefore discarded. The elapsed time between two consecutive fetches is compared with the difference in timestamps of the involved data items. If the former is larger than the latter, the data item is discarded and a next item is checked until the condition is satisfied. Section 2.3 provides a more detailed description of the algorithm.

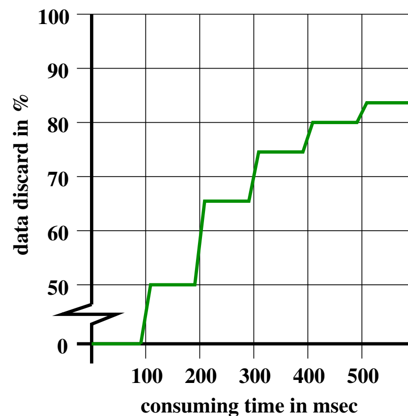

Figure 6.1: Data discard for increasing consuming time.

Consider one component containing a camera and display object. The camera object produces one video frame approximately every 100 milliseconds and sends each frame to the input buffer managed by the display object. If the display object consumes one video frame in less than 100 milliseconds, no frames are discarded. On the other hand, if we increase the

consumer time to values far larger than the 100 milliseconds, more and more video frames are discarded. The test results plotted in Figure 6.1 show how the number of discarded frames increases when the consuming time becomes longer.

6.2 Stream Priority

Stream priorities are implemented by assigning priority values to input buffers. When an input object is configured for multiple streams and data is waiting, the object has a choice of selecting data items from one of its non-empty input buffers. The priority algorithm assures that buffers with higher priority have a larger chance for selection. More specifically, the algorithm adds all the priority values of the non-empty buffers and generates a random positive integer smaller than the cumulative value. Each input buffer is assigned a unique *hit-range* as large as its configured priority value within the bounds of the cumulative value. Consequently, the generated integer falls within one of the hit-ranges and the corresponding input buffer is selected for delivering the next data item.

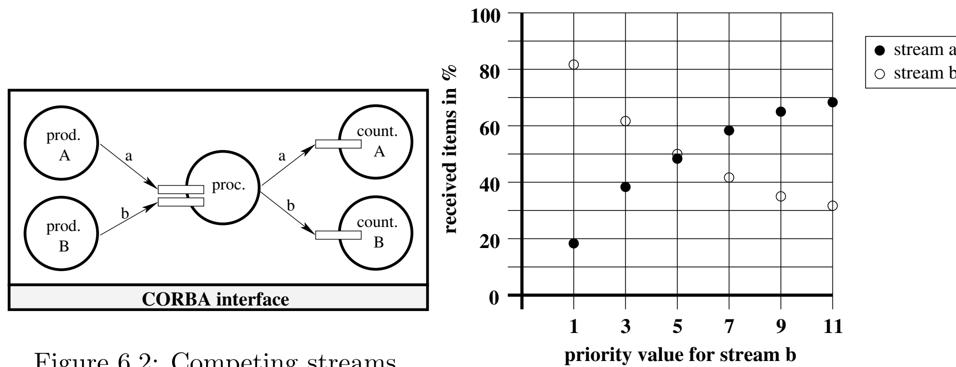

Figure 6.2: Competing streams.

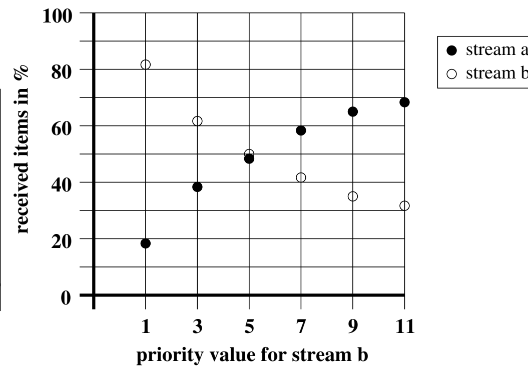

Figure 6.3: Received items for varying priority stream b.

Consider the competing streams in Figure 6.2. The producer objects deliver data at full speed such that both the input buffers belonging to the processor object are non-empty at all times. The processor halts execution after it has forwarded 10,000 data items to the counter objects. The counter objects record the number of data items that have been received and print the result to standard output. These numbers will differ for varying priority values assigned to the streams. Figure 6.3 depicts the results for a fixed priority value of 5 assigned to stream `a` and a varying priority value for stream `b` laid out on the x-axis. The y-axis shows the number of data items received by the counter objects `A` and `B` in percentages of the total number of processed items.

6.3 Distributed Augmented Reality

The distribution of multimedia objects among components has a large influence on the efficiency of configured streams. Objects performing important calculations are typically contained by components running on high-performance machines. Hence, the processing time of data items is reduced and less data items are discarded.

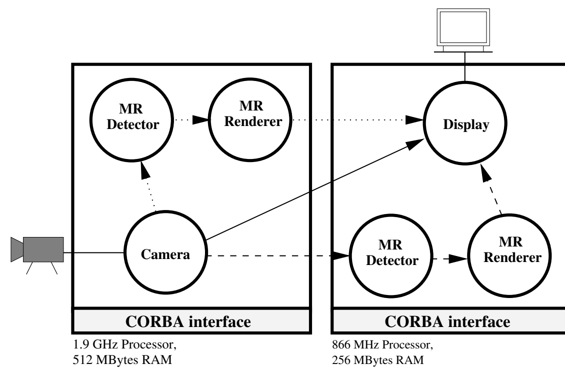

Figure 6.4: Three distributed streams.

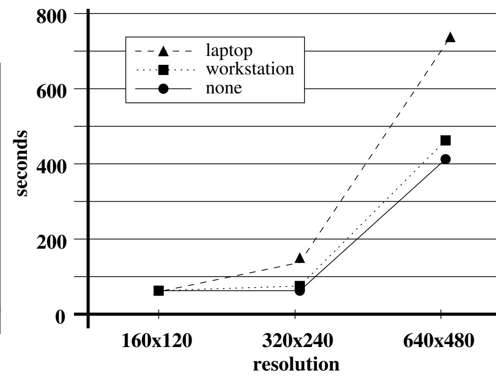

Figure 6.5: Processing time for 2000 frames.

Consider the stream configurations depicted in Figure 6.4 where the left component runs on a high-performance desktop computer and the right component on a low-performance laptop computer. The augmented reality objects for the detection of visual markers and rendering digital images are identical for both components. To evaluate the merits of distribution we measured the performance of three streams; one stream not configured for augmented reality, one utilizing the augmented reality objects contained by the left component and one utilizing the augmented reality objects contained by the right component. The graph in Figure 6.5 shows the time required to display 2000 video frames on the laptop computer when the camera object captures approximately 30 frames per second. The result shows that assigning components containing multimedia objects that perform heavy calculations to high-performance machines increases the throughput of data items considerably.

Chapter 7

Discussion

The augmented reality class library and the example applications, described in Chapter 5, demonstrated the effectiveness of the DCL middleware and the multimedia framework in particular. However, a number of issues still need to be addressed. This chapter discusses the strengths and weaknesses of the current design, and concludes with some suggestions for future experimental and conceptual work.

7.1 Strengths

The multimedia framework provides flexibility to our middleware for the implementation of context-awareness. The CORBA interface defines operations for the reconfiguration of data streams and universal object identifiers managed in proxy objects are easily updated when needed.

Graphical user interfaces can be developed to provide easy means to clients for the (re-) configuration of a set of available components. Because components are designed as self-describing software entities and register themselves at the CORBA Naming Service, clients can retrieve all available components at any time and query for the characteristics of internal objects. In addition, stream information of internal objects can be retrieved and updated. This dynamic information can be used for the construction of an up-to-date graphical user interface representing the configuration of a set of available components.

Reusability of multimedia components and objects simplifies and accelerates the development of augmented reality applications. Off-the-shell objects can be reused in the composition of new components and developed components can be executed on multiple machines at the same time. In addition, multiple components can be configured for one application in order to provide higher-level services. Basically, reusability reduces the required skills and effort from application developers.

Distribution makes it possible to run augmented reality applications on low-performance devices such as PDAs and cellular phones. The component-based design of the multimedia framework enables the developer to assign heavy computation to components that run on high-performance machines. Low-performance devices run light-weight components and delegate heavy computation to stronger machines.

7.2 Weaknesses

The DCL middleware assumes multimedia objects are stateless. Though, when a change of context occurs and universal object identifiers managed by proxy objects are updated, this might be a misplaced assumption. In this case, object state has to move from the old to the new object to maintain the same configuration. We found that objects that interface to hardware devices usually have state.

Distributed streams maintain direct TCP/IP connections for data transmission. When a component crashes, all connected components have to be restarted and reconfigured to restore the old configuration. In addition, TCP/IP connections consume resources and have a slow start-up time. Other data streaming protocols such as RTP, implemented on top of UDP, solve these problems but are very hard to implement.

7.3 Future Work

In future versions, our framework will need support for the transition of objects state to restore the original configuration of a set of components. In addition, RTP has to be considered as a new protocol for data transmission between multimedia components.

Besides extra development work there is a need for exploring new conceptual avenues. It would be very useful to study topics that deal with similar or related issues under a different name such as smart environments, support systems and behavioral context. An in-depth study of such topics provides different views and might help in finding inspiring directions for this middleware.

Chapter 8

Summary

In this master's thesis the multimedia framework that is part of a middleware aiming for fast and easy development of augmented reality applications in ubiquitous environments was described. In this chapter will be the final discussion on how the aim is realized and the merits of this particular solution.

Developers compose an application by specifying services that are implemented by objects contained by multimedia components. Developers use proxy objects that manage a universal object identifier. The middleware automatically updates the identifier referring to an object providing an identical service that is considered the most appropriate object according to current context. Reasoning about the most appropriate object is hidden from the developer by the communication infrastructure. Developers control such reasoning by specifying context policies for the application as a whole. Consequently, developers are not responsible for implementing complex issues involved with context-awareness directly into the application.

The multimedia framework provides a component abstraction as a C++ class. Developers derive from the class and add multimedia objects through its methods. The abstraction hides complex CORBA related features, such as object incarnation and registration with the CORBA Naming Service, which are common for all components. Consequently, the abstraction enables the developer to focus on component composition instead. Moreover, the abstraction defines a standard CORBA interface that is identical for all components resulting in easy usage.

In addition, the framework defines multimedia object abstractions as C++ classes. Object developers specialize these classes and override the appropriate virtual methods for the implementation of custom behavior. Developers configure data streams by invoking methods through the standard CORBA interface belonging to the container component. The abstractions hide details about how the data is streamed between multimedia objects. In addition, the abstractions provide and hide basic functionality such that multimedia objects can be reused in any component without modification. Clearly, reusability speeds up the development of components.

Taken together, hiding context-awareness and providing abstractions for multimedia components and object, provide the developer with the necessary tools for composing augmented reality applications in ubiquitous environments in a relatively fast and easy manner. In addition, the design of the multimedia framework has proved to be effective and easy to use by DCL's middleware developers for implementing automatic reconfiguration of applications in response to context change.

Appendix A

CORBA Basics

This appendix provides a minimal introduction to the Common Object Request Broker Architecture (CORBA) to enable the reader to understand the contents of this master's thesis.

A.1 Intoduction

CORBA is a combination of client-server computing and object-orientated programming. A user-developed client can request for services implemented by a CORBA object. The Object Request Broker (ORB) is the mechanism for handling the interactions from a user-developed client to a CORBA object. The ORB is responsible for finding an object to handle the request, passing the request's parameters to the object, invoking the object's method, and returning the results to the client. Figure A.1 shows a subset of the ORB architecture.

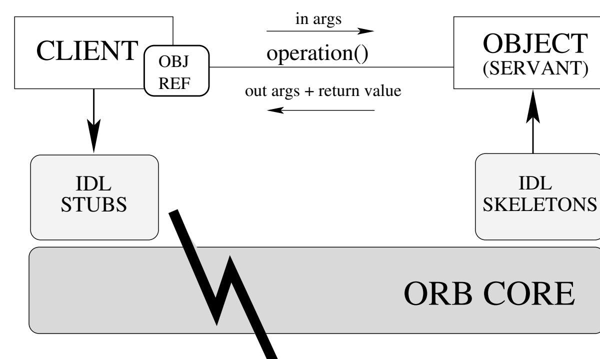

Figure A.1: Subset of Object Request Broker.

When a client invokes an operation on a CORBA object it does not have to be aware where the object implementing the operation is located. The client invokes the operation through a *CORBA object reference* that is used as if it was a reference to a local object.

When developing CORBA objects, the programmer is responsible for writing the interface in CORBA Interface Definition Language (CORBA IDL). These IDL definitions are transformed to the target programming language, in this thesis C++, by a CORBA IDL compiler.

The compiler generates IDL stubs and skeletons that serve as 'glue' between the client and server application, respectively, and the ORB. Programmers derive from the generated skeletons for the implementation of CORBA objects.

A.2 Naming Service

The Naming Service is a standard service for CORBA applications. The Naming Service allows to associate abstract names with CORBA objects and allows clients to find those objects by looking up the corresponding names.

Its role is to allow a name to be bound to an object and to allow that object to be found subsequently by resolving the name within the Naming Service. A server that holds an object reference can register it with the Naming Service, giving it a name that can be used by other components of the system to subsequently find the object. Even though every object in an ORB has a unique reference ID, i.e. a CORBA object reference, from a clients' point of view, it is much easier if there is some directory listing of CORBA objects, so that the client could use a descriptive name to access the object.

A.3 Example

The following example illustrates how a CORBA object is developed. The example object manages a data structure with `long` values. Clients insert and remove values to/from the data structure through the operations defined in the CORBA interface. The example IDL defines one interface called `Numbers` with two operations for insertion and removal of `long` values.

```
interface Numbers {
   void insert(in long n);
   boolean remove(in long n);
};
```

After compilation of the IDL interface, stubs and skeletons are created for the `Numbers` interface. The CORBA object implementation is a derivation of the generated skeleton, which declares pure virtual methods with the same names as the operations defined in the interface. The CORBA object implementation in C++ might look something like this:

```
Numbers_i::Numbers_i()
{
   clearNumbers();
}

void
Numbers_i::insert(long n)
{
   addNumber(n);
}

Boolean
Numbers_i::remove(long n)
{
   bool bSuccess = removeNumber(n);
   return bSuccess;
}
```

A client application is linked with the IDL stubs and a client-side ORB. The client has to obtain a reference to the CORBA object it wants to access, for example through the Naming Service, and invokes methods on this reference. As a result, the ORB finds the target object, passes the method's parameters to the object, invokes the object's method, and returns the result to the client. On the server-side, the CORBA object implementation is linked with the skeleton and a server-side ORB.

Appendix B

Object Execution

This appendix provides the source code for object execution of sources, filters and sinks respectively. Upon object execution, the `Main` method is executed in a separate thread. All types of objects execute a while-loop until a client explicitly stops object execution. Next, a listing of the `Main` method source code and a brief explanation is given for each type of object.

B.1 Sources

```
1.   void
2.   MSource::
3.   Main()
4.   {
5.      LOG_DEBUG1("<Object#%d> Entering MSource::Main()", this->getId());
6.
7.      bool bSuccess = initialize();
8.      setStopped(!bSuccess);
9.
10.     while(!isStopped())
11.     {
12.        m_TimerMutex.Lock();
13.        if(m_nTimer > 0)
14.           m_TimerCond.Wait(calcTimeout(), false);
15.        while(!isConfiguredForStreaming())
16.           m_TimerCond.Wait();
17.        fixStreamIds();
18.        m_TimerMutex.Unlock();
19.
20.        MData* pData = produce();
21.        setStreamIds(pData);
22.        setTimestamp(pData);
23.        getOutport()->send(pData);
24.     }
25.
26.     finalize();
27.
28.     LOG_DEBUG1("<Object#%d> Leaving MSource::Main()", this->getId());
29.  }
```

A source is initialized by a call to the virtual method `initialize` on itself. Consequently, execution is temporarily transferred outside the framework to the developer's code (line 7). The method returns whether it has executed successfully, i.e. whether the object is initialized properly or not. If not, the object is stopped by the framework and will never execute its main-loop (line 8).

Sources have a member variable `m_TimerMutex` that guarantees mutual exclusion on its configured stream identifiers for which the source generates data items. The variable has a built-in timer that allows the thread to be put to sleep for a specified time interval. The timer interval is stored in the member variable `m_nTimer` and is initialized in the object's constructor. The timer interval is used to control the generation rate of data items by sources. For example, if a developer wants to generate 10 data items per second, the `m_nTimer` is to be set to 100 milliseconds. The framework guarantees data items will be generated approximately every 100 millisecond by calling the `calcTimeout` method. If no timer interval is set by the developer, the source produces data items at maximum speed (line 13 to 14).

If the source is not configured for streaming, the `Main` thread is put to sleep by a call to `Wait` on its conditional variable. After a client configures the object for streaming, the `Main` thread is notified and resumes execution (line 15 to 16). Before unlocking the conditional variable, the configured stream identifiers are fixed since after unlocking another thread may change the configuration (line 17).

After unlocking the conditional variable, the pure virtual method `produce` is invoked and execution is temporarily transferred outside the framework to the developer's code that produces a data item (line 20). Then, the fixed stream identifiers and a timestamp are added to the data structure (line 21 to 22) before the data is sent to the source's outport (line 23). The outport is responsible for forwarding the data to its destination object, therefore consulting its stream table.

A source is finalized by a call to the virtual method `finalize` on itself. Consequently, execution is temporarily transferred outside the framework to the developer's code (line 26).

B.2 Filters

```
1.   void
2.   MFilter::
3.   Main()
4.   {
5.      LOG_DEBUG1("<Object#%d> Entering MFilter::Main()", this->getId());
6.
7.      bool bSuccess = initialize();
8.      setStopped(!bSuccess);
9.
10.     while(!isStopped())
11.     {
12.        while(!getInport()->isDataAvailable())
13.           m_InputCond.Wait();
14.
15.        assert(getInport()->isDataAvailable());
16.        MData* pData = getInport()->fetch();
17.        if(pData != 0) {
18.            process(pData);
19.            getOutport()->send(pData);
20.        }
21.     }
22.
23.     finalize();
24.
25.     LOG_DEBUG1("<Object#%d> Leaving MFilter::Main()", this->getId());
26.  }
```

A filter is initialized by a call to the virtual method `initialize` on itself. Consequently, execution is temporarily transferred outside the framework to the developer's code (line 7). The method returns whether it has executed successfully, i.e. whether the object is initialized

properly or not. If not, the object is stopped by the framework and will never execute its main-loop (line 8).

Filters have a conditional member variable `m_InputCond` that is used for synchronizing the `Main` thread with other threads belonging to output objects. When there is no data available in one of its input buffers, the `Main` thread is put to sleep until it receives data from another (output) thread. When data arrives, the `Main` thread is notified and resumes execution (line 12 to 13).

After being assured data is waiting, the data is fetched from its inport (line 15 to 16). Then, if the data is valid, the pure virtual method `process` is invoked and execution is temporarily transferred outside the framework to the developer's code that processes the data (line 17 to 18). Finally, the processed data is send to the filter's outport that is responsible for forwarding the data to its destination object, therefore consulting its stream table (line 19). Note that the condition in line 17 will fail if and only if the data that arrived in its input buffer is obsolete and therefore discarded.

A filter is finalized by a call to the virtual method `finalize` on itself. Consequently, execution is temporarily transferred outside the framework to the developer's code (line 23).

B.3 Sinks

```
1.  void
2.  MSink::
3.  Main()
4.  {
5.     LOG_DEBUG1("<Object#%d> Entering MSink::Main()", this->getId());
6.
7.     bool bSuccess = initialize();
8.     setStopped(!bSuccess);
9.
10.    while(!isStopped())
11.    {
12.       while(!getInport()->isDataAvailable())
13.          m_InputCond.Wait();
14.
15.       assert(getInport()->isDataAvailable());
16.       MData* pData = getInport()->fetch();
17.       if(pData != 0) {
18.          consume(pData);
19.          if(m_bDelete)
20.             delete pData;
21.       }
22.    }
23.
24.    finalize();
25.
26.    LOG_DEBUG1("<Object#%d> Leaving MSink::Main()", this->getId());
27. }
```

Sinks behave identically to filters until line 16. From there on, the pure virtual method `consume` is invoked and execution is temporarily transferred outside the framework to the developer's code that consumes the data (line 18). Finally, if the member variable `m_bDelete` is set by the object developer in the sink's constructor, the framework deletes the data from memory. Otherwise, the object developer himself is responsible for releasing the data (line 19 to 20).

A sink is finalized by a call to the virtual method `finalize` on itself. Consequently, execution is temporarily transferred outside the framework to the developer's code (line 24).

Index

application composer, 3
augmented reality, 1
augmented reality class library, 35

basic stream, 10, 24
broadcasting stream, 10, 25

communication infrastructure, 3
competing stream, 11, 27
component, 7
component interface, 15
configuration manager, 4
context database, 4
CORBA, 47
CORBA component reference, 8
CORBA IDL, 47
CORBA interface, 7, 8, 14
CORBA Naming Service, 48
CORBA ORB, 47
cyclic stream, 10, 24

data, 11, 28
data discard, 12, 39
data streams, 9, 23
distributed stream, 10, 25

filter, 9, 20, 52

IDL, 47
inport, 9, 19
input object, 9
input type, 9
Interface Definition Language, 47

location policy, 4

middleware, 3
multimedia component, 7, 14
multimedia data, 11, 28
multimedia framework, 3
multimedia object, 7, 8, 17

Naming Service, 48

object, 7, 8
object identifier, 9
Object Request Broker, 47
ORB, 47
outport, 9, 19
output object, 9
output type, 9

performance policy, 4
pervasive computing, 1
proposed middleware, 3
proxy object, 4

service, 8
sink, 9, 20, 53
source, 8, 20, 51
standard CORBA interface, 8
state interface, 17
stream identifier, 23
stream interface, 16
stream priority, 11, 27, 40
streams, 9, 23

TCP client, 26
TCP server, 26

ubiquitous computing, 1
universal object identifier, 9

Bibliography

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Flissides: *Design Patterns, Elements of Reusable Object-Orientated Software*, Addison-Wesley Publishing Company (1995), ISBN 0-201-63361-2.

[2] Bjarne Stroustrup: *The C+ Programming Language - Special Edition*, Addison-Wesley Publishing Company (2001), ISBN: 0-201-70073-5.

[3] Michi Henning, Steve Vinoski: *Advanced CORBA Programming with C++*, Addison-Wesley Publishing Company (1999), ISBN 0-201-37927-9.

[4] Sai-Lai Lo, David Ridoch, Duncan Grisby: *The omniORB version 3.0 User's Guide*, AT&T Laboratories Cambridge (2002).

[5] Christopher J. Lindblad, David L. Tennenhouse: *The VuSystem: A Programming System for Compute-Intensive Multimedia*, Massachusetts Institute of Technology, Cambridge MA 02139.

[6] prof. dr. ir R. L. Lagendijk: *http://www.ubicom.tudelft.nl/*

[7] Martin Bauer, Bernd Bruegge, et al.: *Design of a Component-Based Augmented Reality Framework*, Technische Universitat Munchen.

[8] Schulzrinne, Casner, Frederick, Jacobson: *RTP: A Transport Protocol for Real-Time Applications*, Columbia University.

[9] Tatsuo Nakajima: *Experiences with Building Middleware for Audio and Visual Networked Home Appliances on Commodity Software* Department of Information and Computer Science, Waseda University.

[10] Andrew P. Black, Jie Huang, et al.: *Infopipes: an Abstraction for Multimedia Streaming*, Department of Compter Science & Engineering, Oregon Health & Science University, ...

[11] Diego Lopez de Ipina, Paulo R.S. Mendonca, Andy Hopper: *TRIP: a Low-Cost Vision-Based Location System for Ubiquitous Computing*, Laboratory for Communications Engineering, University of Cambridge, UK; Fallside Laboratory, University of Cambridge, UK; AT&T Laboratories Cambridge, UK

[12] G.D. Abowd, E.D. Mynatt, "Charting Past, Present, and Future Research in Ubiquitous Computing", ACM Transaction on Computer-Human Interaction, 2000.

[13] ARToolkit, http://www.hitl.washington.edu/people/grof/SharedSpace/Download/ARToolKitPC.htm.

[14] R.T. Azuma, "A Survey of Augmented Reality", Presence: Teleoperators and Virtual Environments Vol.6, No.4, 1997.

[15] M. Weiser, "The Computer for the 21st Century", Scientific American, Vol. 265, No.3, 1991.

[16] G.Banavar, J.Beck, E.Gluzberg, J.Munson, J.Sussman, D.Zukowski, "Challenges: An Application Model for Pervasive Computing", In Proceedings of the Six Annual International Conference on Mobile Computing and Networking, 2000.

Design of a Component-Based Augmented Reality Framework, The Second IEEE and ACM International Symposium on Augmented Reality, 2001.

[17] G.S.Blair, et. al., "The Design and Implementation of Open ORB 2", IEEE Distributed Systems Online, Vol.2, No.6, 2001.

[18] Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vicente, Daniel Villela, "A Survey of Programmable Networks", ACM SIGCOMM Computer Communications Review, Vol.29, No.2, 1999.

[19] A.K.Dey, G.D.Abowd, D.Salber, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications", Human-Computer Interaction, Vol.16, No.2-4, 2001.

[20] Nathaniel I. Durlach and Anne S. Mavor. "Virutal Reality : Scientific and Technological Challenges" National Academy Press (1995). ISBN 0-309-05135-5

[21] N. Gershenfeld, "When Things Start to Think", Owl Books, 2000.

[22] Steven Feiner, Blair MacIntyre, and Doree seligmann. "Knowledge-based Augmented Reality", Communications of the ACM 36, 7 (July 1993) , 52-62

[23] Andy Hopper, "Sentient Computing", In *the Clifford Paterson Lecture*, volume 358, pages 2349-2358, Phil. Trans. R. Soc. Lond., September 1999

[24] Anantha R. Kancherla, Jannick P. Rolland, Donna L. Wright, and Grigore Burdea. "A Novel Virtual Reality Tool for Teaching Dynamic 3D Anatomy", Proceedings of Computer Vision, Virtual Reality, and Robotics in Medcine '95 (CVRMed '95) April 1995.

[25] Diego Lopez de Ipina and Sai-Lai Lo, "LocALE: a Location-Aware Lifecycle Environment for Ubiquitous Computing", In Proceedings of the 15th IEEE International Conference on Information Networking (ICOIN-15), 2001.

[26] T.Nakajima, "System Software for Audio and Visual Networked Home Appliances on Commodity Operating Systems", In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, 2001.

[27] T.Nakajima, H.Ishikawa, E.Tokunaga, F. Stajano, "Technology Challenges for Building Internet-Scale Ubiquitous Computing", In Proceedings of the Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems, 2002.

[28] T.Nakajima, "Experiences with Building Middleware for Audio and Visual Netwoked Home Appliances on Commodity Software", ACM Multimedia 2002.

[29] OMG, “The Common Object Request Broker Architecture: Architecture and Specification”, October 1999

[30] OMG, “Final Adopted Specification for Fault Tolerant CORBA”, OMG Technical Committee Document ptc/00-04-04, Object Management Group (March 2000).

[31] C.Pinhanez, “The Everywhere Display Projector: A Device to Create Ubiquitous Graphical Interfaces”, In Proceedings of Ubicomp’01, 2001.

[32] K.Raatikainen, H.B.Christensen, T.Nakajima, “Applications Requirements for Middleware for Mobile and Pervasive Systems”, Mobile Computing and Communications Review, Octorber, 2002.

[33] Jun Rekimoto, “Augmented Interaction: Interacting with the real world through a computer” , HCI International, 1995.

[34] Mihran Tuceryan , Douglas S. Greer, Ross T, et. al., “Calibration Requirements and Procedures for Augmented Reality”, IEEE Transactions on VIsualization and Computer Graphics 1, 3 (September 1995), 255-273