UNIVERSITY OF OSLO
Department of Informatics

Multimedia Stream Adapter for Object-Oriented Middleware

Cand Scient thesis

Doru-Catalin Togea

19th November 2004

Acknowledgments

First of all, I would like to thank God for the fact that I could finish this thesis. He knows – life isn't always easy. Especially thanks for all those blessings which I have not noticed to have come from Him, but which I have benefited from and which I have probably also enjoyed in spite of my ignorance. I hope I will be able to thank Him one day, face to face.

Secondly, I would like to thank my mother Cornelia Togea (Sr.) and my sister Cornelia Togea (Jr.) for helping me to the uttermost of their abilities. Thank you mother, especially for being able to come back "home", for so long.

Thirdly, I would like to thank my two supervisors, Thomas Plagemann and Tom Kristensen for everything they have done for me: tutoring me, giving encouragements, being very, very patient with my procrastinations and so on. Especially thanks to Thomas Plagemann who has helped me to get a much needed extra period of three weeks at the very end of my work with this thesis.

Many thanks also to my sensor, Anders Andersen, for being willing to grade this thesis in a much reduced period of time. Many thanks also to the administrative staff of the Institute for Informatics who have granted me this postponement.

Fourthly, I would like to thank prof. Stein Krogdahl for answering my questions about parsing and grammar conversions. Many thanks also to f.aman. Carsten Griwodz who has been so helpful when I could not figure out some obstinate bugs in my programs. Some other individuals have also shared with me fromtheir insight.

Many thanks to Ovidiu-Valentin Drugan, for helping out with the printing of this thesis ;)

Finally, I would like to thank my good friend, Karmen Palts, for being such a good friend to me. No one has thought me more about love then you.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Why a multimedia stream adapter?

The work presented in this thesis, embodied in the concept of a *Multimedia Stream Adapter* ***(MSA)***, finds its motivation in the efforts made to join two separate areas of development in computer science, which have evolved much during more then a decade now.

First of all, we have the area of *Open Distributed Processing* ***(ODP)*** which has established itself during the last years as a valid and useful computing *paradigm*. ODP has been *incarnated* in framewoks for distributed computing. Among these, CORBA is probably still the one most widely known and used. Recently, however, several other frameworks have evolved and we have seen a tendency for these newcomers to steal more and more of CORBA's share in the market. We think of frameworks like SOAP [30], .NET [17] and ICE, just to mention a few. We are particularly concerned with ICE in this thesis.

The second area of concern for this thesis is the area of multimedia processing. For many years already, computer hardware designed specifically to process multimedia data has become both powerful and cheap. Of course, there still is the hardware on the cutting edge of development which can be quite expensive, like professional video editing cards, but in general, there is a lot of affordable hardware for multimedia processing available for today's computers. Regular video cards are quite powerful and have a generous amount of dedicated video memory. There are MPEG-2 and MPEG-4 decoders, myriads of video editing cards and quite a few USB and FireWire based external devices, like digital camcorders, webcams and digital cameras.

The developments in these two areas have motivated computer science researchers all over the world to find ways to combine them in frameworks for *Multimedia Open Distributed Processing* ***(MODP)***. The goal is to create systems which provide both 1) the benefits of distributed computing, like *location transparency* and support for hardware and operating system *het-*

erogeneity, and 2) support for multimedia data. The basic challenge posed by trying to unite multimedia processing and distributed computing is that the latter has been designed with *discrete* data in mind and it does not automatically offer the same support for *continuous* data, which is typical for multimedia applications.

1.2 Project context

Initially, this thesis was defined as part of the *Multimedia Middleware for Low Latency High Throughput Environment* ***(MULTE)*** project, run as a cooperative effort between the Center for Technology at Kjeller and the Institute of Computer Science at the University of Oslo. The goal of the MULTE project is to specify requirements for and make an implementation of the MULTE-ORB.

The main components of MULTE-ORB are the *Chorus Object-Oriented Layer* ***(COOL)*** [29, 28] and the *Dynamic Configuration of Protocols* ***(Da CaPo)*** system [5, 23].

The first of these main components, COOL, is a CORBA compliant ORB implementation, based on Chorus, a micro-kernel based operating system. At the time when the work with this thesis began, it became apparent that Chorus did not offer all those features which the MULTE project's researchers hoped to find in it. Therefore, the project members looked for alternatives to COOL. TAO-ORB and OmniORB have been considered.

The second main component of the MULTE-ORB is Da CaPo. Da CaPo is a system designed to allow the creation of highly customizable and run-time adaptable application specific protocols. The initial implementation of Da CaPo [5, 23] has been integrated into COOL in the early stages of the MULTE project [15]. When the work with this thesis began, other project members were already working on reimplementing Da CaPo and were looking for a better way to integrate the new implementation into COOL, or later into COOL's replacement.

Because of working assignments outside of the MULTE project, both on the part of this author and on the part of other project members, the work presented in this thesis has continued on its own for quite a while.

1.3 Thesis goals

In general terms, the goals of this thesis are to:

1. analyze in which ways ***streams*** and ***flows*** of data can be treated as first class objects within object-oriented middleware, and to

2. provide an implementation for a MSA based on the results of point 1 above.

The goals set for this thesis have been influenced by the fact that there has been a shifting away from a close connection with the MULTE project to an independent work. We distinguish three *stages* in the life span of this thesis, and each stage has its own goals:

1. **MSA for MULTE-ORB**.
   To begin with, our goals were 1) to present an analysis of how the MSA could contribute to fullfill one or several of the requirements identified for the MULTE-ORB in [21], and 2) to implement some of these requirements by means of implementing the MSA, all in the context of integrating the new implementation of Da CaPo into MULTE-ORB.

2. **Da CaPo implementation**.
   Since the reimplementation of Da CaPo has not been finished by the time we needed it, we have redefined part of the goals of this thesis to consist in the implementation of a Da CaPo core. The implementation of a complete Da CaPo system is far too much work for a master's thesis, regardless of wheather we attempt to integrate Da CaPo into an ORB or not.

3. **MSA for another object-oriented middleware**.
   Since COOL was rejected as the ORB for the MULTE project and the work on MULTE-ORB had been temporarily discontinued, before other MULTE-ORB project members decided which other ORB they wanted to base their work on, we had to make a decision on our own. The goal became then to implement a MSA for another object-oriented middleware platform.

Stage 3 above completely overshadows stage 1, but only as far as this thesis' goals are concerned. As we see in Chapter 4, we still consider the requirements posed to a MSA for the MULTE-ORB as useful guidelines for the implementation of a MSA for another object-oriented middleware platform.

By eliminating the goals derived from stage 1 above, and by being a little more specific, we can summarize the goals of this thesis, as follows:

1. **Da CaPo core implementation.**
   Some basic ideas for the new Da CaPo implementation have been developed before this thesis has drifted apart from the rest of the MULTE project. Our implementation builds on these ideas.

2. **Establish Requirements for an middleware based MSA.**
   We use the requirements identified for the MULTE-ORB as a starting point for the requirements we identify for the ICE based MSA presented in this thesis.

3. **MSA implementation.**
   We provide an implementation of the MSA, based on the requirements defined in point no. 2 above.

4. **The integration of Da CaPo and the MSA.**
   Show how our Da CaPo implementation and the MSA can be integrated into a common system.

1.4 Research method

The methodological approach used in this thesis has been an iterative cycle of 1) literature studies, 2) analysis and design and 3) implementation and testing. This approach follows closely the description of the three paradigms, or processes, used to define the discipline of computing in [2].

Since *"in computing the three processes are so intrinsically intertwined that it is irrational to say that any one is fundamental"* [2], we present them all before we pinpoint where our thesis belongs mostly.

- **Theory**.
  Theory is based on mathematics and consists of four iterative steps taken to develop a coherent, valid theory: 1) make definitions, 2) hypothesize about your definitions, 3) prove which of your hypothesis are true and 4) interpret the results.

- **Abstraction**.
  Abstraction (also called *modelling* or *experimentation*) is based on the experimental scientific method, and it also consists of four iterative steps taken in the analysis of a phenomenon: 1) hypothesize, 2) construct a model and make predictions, 3) perform experiments and collect data and 4) interpret the results.

- **Design**.
  Design is based on established engineering methods and consists of four iterative steps taken to construct a system which solves a given problem: 1) establish the requirements, 2) establish the specifications, 3) design and implement the system and 4) test the system.

In this thesis we are not concerned with establishing new theoretical results. The theoretical background we use is already established in the work behind the systems we use. As a special case we mention the FIDL specification language, which is introduced in Section 3.3, and which has a well defined theoretical background. The same applies for ICE and Da CaPo.

As far as abstractions are concerned, we do not introduce new ones in this thesis. We only give a new application to abstractions made by others, such as the concepts of streams and flows, in general, and the object adapter of an

ORB. We also employ the applications given by others to abstractions such as the *Remote Procedure Call* ***(RPC)*** concept, which is used in a *callback* manner in this thesis[1].

We consider that this thesis belongs mostly to the design paradigm, because we attempt to implement a new *specific instance* [2] of the relationships established by others between the abstractions specific to multimedia processing (streams and flows) and object-oriented middleware platforms (like ICE).

The research method employed in this thesis mostly resembles therefore the iterative steps of the design paradigm. In addition to those steps, we had to also engage in an iterative step of relevant literature study, until we achieved an understanding of the theory and of the abstractions employed which is detailed enough to form the knowledge base of our implementation.

1.5 Contributions

The authors of [2] state that *"The fundamental question underlying all of computing is, What can be (efficiently) automated?"* There are many things we can be efficiently automated by computer systems. We consider that also the work presented in this thesis is a direct answer to this question, as we present an alternative way to automate the process of distributed programming with streams of multimedia data.

Our contributions are:

- The implementation of a Da CaPo core. Da CaPo is a system for the design of application specific communication protocols.
- The implementation of a MSA, which is an ORB-like system which allows the programmer to refer to streams of data as first class C++ objects in distributed programs.
- FIDL++, which is a slightly extended version of the FIDL specification language.
- fidl, a compiler which processes FIDL++ specifications.
- A set of demo applications which show how Da CaPo and the MSA can be used.

1.6 Terminology

1.6.1 Client and server

The terms *client* and *server* can be confusing at times. This happens especially in situations where an application plays the role of a client toward

[1] This is discussed in Section 5.2.2

some applications and the role of a server toward other applications.

In this thesis we give a general meaning to these terms. In a distributed multimedia application, by ***client*** we always mean that side of the application where the multimedia data is consumed and by ***server*** that side of the application where the data is produced.

For a symmetric application, like a video conferencing application, each participant is both a client and a server.

In other scenarios, like a surveillance camera application, we call the application side which interfaces the camera *the server*, and the application side where the images are displayed *the client*. We use this terminology, in spite of the fact that, as we shall see, the two sides of a surveillance camera application can play both roles toward each other, in the process of transmitting the images from the camera device to the screen.

1.6.2 Demo applications

We have implemented three demo applications in this thesis. However, one of them, the MSA demo application, is the most proeminent among them. Therefore, we often refer to it as only "our demo application. When we mean one of the other demo applications, we will explicitly make that clear in the text.

1.6.3 FIDL vs. FIDL++

In this thesis we make very small additions to the FIDL specification language. Because of lack of a better name, we call "our" new language FIDL++. However, the two languages are completely the same, as far as their ability to specify QoS requirements is concerned. Most times we do not make a distinction between them in this thesis. The major contribution of this thesis, in regard to FIDL, is the compiler we have implemented for our version of the language, not the small additions we have made.

1.7 The structure of this thesis

Chapter 1 gives on overview of the key issues which we deal with in this thesis and presents the goals which we had for this thesis.

Chapter 2 presents general background material which is important for an understanding of the issues which we deal with and of the choices we make in this thesis.

Chapter 3 present very specific background material, which is directly relevant to this thesis, but which is not our contribution.

Chapter 4 gives first an overview of the requirements posed to our MSA, and continues then with an analysis of the requirements which helps us to chose the platforms and tool which we use in the reminder of this thesis.

Chapter 5 presents some of the most relevant conclusions this author has come to, by means of the literature studies made. These conclusions allow us to further refine the choice of platforms and tools made in the previous chapter.

Chapter 6 presents our design and implementation of Da CaPo.

Chapter 7 presents the interpretation we have given to the FIDL specification language, in order to be able to implement a compiler and code generator for it.

Chapter 8 presents the process of code generation executed by our compiler when FIDL++ files are parsed, and describes the code which is generated for a MSA based application.

Chapter 9 gives a description of a MSA based application's run-time.

Chapter 10 presents the streaming API supported by our MSA.

Chapter 11 presents the three demo application implemented in this thesis.

Chapter 12 presents our evaluation of the work done in this thesis.

Chapter 13 presents our conclusions and suggestions for further work.

Appendix A presents a stepwise recipe for how to build an MSA based application from scratch. It uses our MSA demo application as an example.

Chapter 2

General background

In this chapter, we present a few systems and frameworks which are of general interest to this thesis. We begin with general multimedia issues which are related to our work, in Section 2.1. In Section 2.2 we present am overview of key issues in ODMP. Section 2.3 gives a presentation of the *Reference Model of Open Distributed Processing* ***(RM-ODP)***, which is an important referential framework. We end this chapter by presenting CORBA and the A/V Streams Specification, in Sections 2.4 and 2.5.

2.1 General multimedia issues

2.1.1 Data classification

In terms of how data is perceived in a computer program, we can classify it in several ways:

- ***Discrete*** or ***continuous*** data.
  By discrete data we mean any piece of data. The context decides what is the natural granularity of a discrete piece of data. In a text processing application a *file* can be an example of a discrete piece of data in one context, and a *paragraph*, a *sentence* or a *word* can be natural pieces of discrete data in other contexts. In an image processing application, an image can be the natural discrete piece of data in some contexts, or an 8x8 area of pixels of an image can be a piece of discrete data, as in the case of JPEG compression operations.

  By continuous data we mean a sequence of discrete pieces of data. The sequence can be *short*, consisting of only a few discrete pieces of data, or *long*, consiting of a great number of discreate data pieces. The sequence can also have a *determined length*, like the definite number of frames in a video file, or it can have an *undetermined length*, as in the case of the images provided by a surveillance camera. Even though the data is actually discrete from the point of view of what it is made of, it

must be treated as continuous in applications which look at sequences of discrete pieces of data as a whole.

- ***Simple*** or ***combined*** data.
  By simple data we mean data which is of only one type when it is presented to a program. Examples are text, images, sound, etc. By combined data we mean data which consists of several interwoven simple types of data. Examples are *Digital Video* ***(DV)*** data or data in any of the many MPEG file formats. In both of these examples, audio data is stored together with video data, and neither of them can be accessed on its own until specialized code breaks them appart and reconstructs the audio and the video tracks.

- ***Compressed*** or ***uncompressed (raw)*** data.
  By uncompressed data we mean data as it is when it is generated. Text which is typed in a text editor is uncompressed. It can be compressed later. Images provided by the optical sensors in a digital camera are uncompressed. They can be compressed by hardware or software in the digital camera, for instance into JPEG images, before they are uploaded to a computer, or they can remain uncompressed. After uploading raw images, a computer can either retain them in their uncompressed format, or it can compress them itself, to one of the many available compressed image formats.

  Compression algorithms can be *lossy* or *lossless*. Data compressed with a lossless compression algorithm can always be decompressed to the exact original from which it has been compressed from. Lossy compression algorithms provide usually a better compression ratio than the lossless ones, but when decompressing the compressed data, we can only obtain an approximation of the original data.

  Another, and for this thesis more interesting, issue in regard to compressed data is the ***intrinsic complexity*** of the compressed data. Many compression schemes base their effectiveness on the observation that there is much redundancy in the original data. A typical example is a sequence of frames of a video of some static scenery. In this example, most frames in the sequence vary very little one from another. Therefore, an effective way to compress such a sequence of frames is to store only one *reference* copy of the frames and the *differences* between each of the other frames and the reference frame. The frames for which only the difference is stored are called *referencing* frames. Upon reconstruction of the original frame sequence, the referenced frame must be combined with the the difference of each referencing frame. This imposes an intrinsic dependency between the elements of the compressed data. The referencing frames can not be reconstructed without the referenced frame. In some compression schemes, like MPEG-4, it

is common that referencing frames[1] do refer to both prior and posterior referenced frames, thus creating even greater intrinsic complexities within the data.

Data with intrinsic complexities greatly impairs the freedom of an application to manipulate the individual discrete pieces of data, as the data must be decompressed first. It is therefore impossible to manipulate the individual discrete pieces of data, for instance during transportation.

2.1.2 Types of multimedia data

The term "multimedia" is very inclusive, and, in the context of distributed multimedia programming, leads us to think of the availability of a plenitude of data types to distributed applications. This generality of the word's semantics is not harmful in any way, but, in practice, we see that there are not that many types of media after all. Very often "multimedia applications" are not providing anything else to users than ***video*** and the associated ***audio*** tracks.

In some scenarios we can also see the need for other types of data. In a Video-On-Demand presentation, the subtitling could be sent to the viewer application(s) as a separate stream of data of type *text*. In an advanced video conferance scenario using smart boards it would be necessary to send the state of the smart boards and the position of the pointing device back and forth between the end-points participating to the conference. In a medical consultation scenario, say remote assistance from a specialist in a case of surgery, the patients measurements of heart rate, blood pressure, temperature, etc. should also continually be sent to the assisting specialist, say five times every second, together with a video and audio presentation of the surgery. These measurements would typically be numerical values and would have a data type corresponding to the traditional *int*, *long* or *float* types common in programming languages.

In the average multimedia application the preponderance of the video and audio data largely exceeds the other types of data, first of all because these two types of data are "always" present, and secondly, they also are the most voluminous of the data types in use in multimedia applications today. It is therefore imperative for a framework for multimedia distributed programming to provide very good support for the video and audio data types. Of course, in order to be more then just (another) video player, a framework for multimedia processing must give general support to any kind of data.

In the rest of this thesis, we use the term ***multimedia data*** to mean any kind of data, but we keep in mind that in most practical situations

[1] Also called *frames*.

this means primarily video and audio, as it also is the case with the demo applications of this thesis.

2.1.3 Streams and flows

The concepts of ***stream*** and ***flow*** have been used for a long time now to model how continuous data is experienced by applications, when they need to look at it at a larger granularity than the individual discrete unit of data of which the stream is made of.

We define a flow to mean a sequence of discrete units of data of the same type. Often we need to refer to the fact that a flow is *flowing*, which means that the discrete pieces of data which make up the flow are traveling from one point to another. An example of a flow is the sequence of images sent by a surveillance camera.

We define a stream to be a collection of one or several flows which are somehow related to each other within an application. An example of a stream would be the set of an image flow from a surveillance camera and the flow of sound from the microphone connected to the surveillance camera. If all the flows within a stream are flowing, we will say that the stream is *streaming*.

These are not the only ways in which the stream and flow concepts have been used by the ODMP research community. One alternative is the way they are defined and used in FIDL, as described in Section 3.3.

2.1.4 QoS

Multimedia data is often characterized by many parameters. Images, for instance, can be of different sizes, use different encoding schemes[2] like the *YUV 4:2:0 Planar* ***(YUVp420)*** or the *Reg-Green-Blue-Alfa* ***(RGB32)*** formats, and they can be rendered by means of different *colour palettes*, which can contain a different number of available colours. The exact value of such parameters determines how a user experiences the data when it is presented to him or her.

In order to be able to formally reason and speak about the *quality* of multimedia data, the research comunity has tried to find ways to express ***QoS requirements*** and ***QoS guarantees***. These are opposite concepts, in the sense that *QoS requirements* are expected from an application and *QoS guarantees* are provided to an application.

A comprehensive treatment of the subject of QoS is given in [1]. There it is specified that QoS management *functions* can be grouped into:

- **Static** functions, like QoS *specification*, *negotiation*, *resource reservation*, *admission control*, and
- **Dynamic** functions, like QoS *monitoring* and *renegotiation*.

[2] An encoding scheme is how the data is represented and/or stored in computers.

Since QoS *"is apparent in all layers in a communications architecture, but is 'viewed' differently by each layer"* [4], the QoS concept can be addressed at any of the layers shown in the two subfigures of Figure 2.1. We see here that the general QoS layering of Subfigure b) can be further refined as in Subfigure a). By *DPE* in Subfigure a), [4] means *Distributed Programming Environment*, like CORBA ORBs.

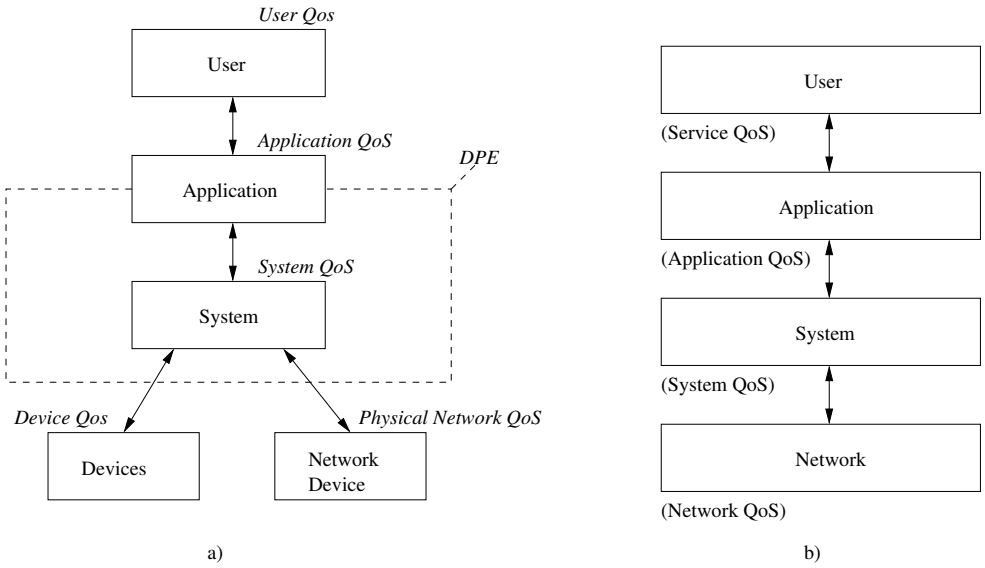

Figure 2.1: *QoS layers: a) [4], b) [31]*

In general, QoS requirements are more *generic* at higher levels of specification and more *specific* at lower levels of specifications. The specification given in parameters typical to one level must be translated to specifications given in parameters typical to other levels. For instance, at user level we describe the QoS in terms like *very good*, *good*, *acceptable*, *least acceptable*, and so on. In a video conference application's case, "very good" could mean, at the application level, VGA sized images at a framerate of 15 or higher with stereo sound, while "least acceptable" could mean sound only, mono, at 8000 samples per second, which would be telephone quality. At the network level, all higher level QoS requirements will have to be mapped to very specific parameters like delay <= 50ms. Table 1 of [21] gives an overview of possible lowlevel QoS parameters.

QoS specification for video data

There are basically three parameters by which video can be adjusted to different *levels of QoS*. These are the *frame rate*, the *resolution* and the *color depth*.

If the video data is compressed in a way which generates intrinsic dependencies between the individual images, it is difficult to easily adjust the frame-rate parameter, because the images are not independent of each other any more and can therefore not be dropped arbitrarily.

In this thesis, we process raw video data, in which there are no other dependencies between the individual images than their inherent timely ordering. The granularity at which we process video data is one image per *packet*, as we will see.

QoS specifications for audio data

Conceptually, the parameters which can be used to specify QoS requirements for sound are the *samplerate*, the *samplesize* and the number of *channels*.

Since a sample of even the best quality of sound takes only a few bytes of storage, it is not feasible to treat samples one by one, on a sample per *packet* basis. In this thesis, we generate sound packets which always contain a 1 second's worth of sound. This is good enough for the sake of our demonstrations, but the amount of sound in a packet can easily be adjusted. What we will focus on is how the size of the packet varies with the three parameters mentioned above.

2.2 Key issues in ODMP

Middleware has emerged as a central architectural component in supporting distributed applications and services. The role of middleware is to present a higher level programming paradigm for application writers (typically object-oriented or, more recently, component-based) and to mask out problems of heterogeneity and distribution [21].

Combining ODP middleware and multimedia into ODMP has proven to raise many challenging issues. They arise mainly because, according to its original intent, ODP middleware was developed to provide support for distributed, but *not* continuous patterns of interaction. Examples of this kind of interaction are the *Remote Procedure Call* ***(RPC)*** and the *Remote Method Invocation* ***(RMI)*** mechanisms. Section 1.3.2 of [1] identifies four main areas where multimedia challenges traditional ODP frameworks. They are:

1. **Support for continuous media.**
   Multimedia data is often of a continuous nature and can be perceived as streams and flows of data. The characteristics of streams and flows of data challenge the established middleware frameworks, as operations on them can not be easily implemented by means of RPCs or RMIs. Among other things, streams' data can have unpredictable durations,

and there can be many dependencies among the flows within a stream, such as requirements for synchronization.

2. **Real-time synchronization.**
   The continuous character of most multimedia data imposes requirements for synchronization to applications. real-time synchronization requirements can be classified as *hard* or *soft* requirements or as *inter-flow* or *intra-flow* requirements. This subject is treated in detail in [27]. An example of inter-flow synchronization is the need for lip-synchronization between an audio and a video flow, in an application like video conferencing.

3. **Quality of service management.**
   Synchronization dependencies, as well as other peculiarities of multimedia data are expressed by means of QoS requirements. This is a feature which distributed multimedia applications share with other applications, like real-time and safety critical applications. However, in the light of this ODMP discussion, research has shown that 1) ODP middleware does not support the notion of QoS to any significant extent, 2) the requirements imposed by the new application domains coming into existence are perceived to be "invading" the ODP frameworks, among other things because they are so diverse, and 3) future application domains with new requirements may appear, [1, 15, 5, 23]. Therefore, new generation of middleware is must be both *flexible* and *extendable* in addition to supportive of the QoS concept.

   QoS management encompasses a number of different functions, which can be grouped into:

   (a) **static aspects** like QoS *specification*, *negotiation*, *resource reservation*, *admission control*, and

   (b) **dynamic aspects** such as QoS *monitoring* and *renegotiation*.

   QoS managements is required at a number of different levels in the middleware, like application, transport, network, and operating system, with appropriate mappings between the various layers. In distributed systems, QoS support should be *end-to-end*, extending from the information source to the information sink, thus requiring a certain amount of coordination between the end-points and the network infrastructure.

4. **Multiparty communications.**
   Some distributed multimedia applications, like video-conferences and long-distance-learning applications, are concerned with interactions between dispersed groups of users. It is now recognized that this requires explicit support from the underlying distributed systems platform. The

authors of [1] identify the need for a programming model which supports multiparty communications for both discrete and continuous media types. This model must facilitate the management of groups of users, providing support for operations like the joining and leaving of groups at run-time. Providing this kind of support is complicated by the heterogeneity in hardware, which potentially exists among the participants of such shared multimedia sessions. Different participants might impose different QoS requirement.

2.3 RM-ODP

RM-ODP [13, 1] is probably the only framework for distributed computing which has been standardized. As such it is a referential framework which facilitates the comparison of ODP platforms to each other. RM-ODP does not prescribe *how* to implement an ODP platform, but provides *abstractions* which an implementation can be based on. RM-ODP provides 5 levels of abstraction, from which an ODP platform can be viewed. They are also called *viewpoints*. For each level a set of concepts, structures and rules is provided. Together they form a "language" for the given level, in which the platform can be described. The five viewpoints are the ***enterprise***, the ***information***, the ***computational***, the ***engineering*** and the ***technology*** viewpoints. For our work, the engineering and the technology viewpoints are the most relevant, so they are the only ones we present here.

2.3.1 The computational view

In this viewpoint, the functionality of a distributed system is specified in a distribution-transparent manner, by means of *computational objects*. The interaction between computational objects is encapsulated into so called *binding objects*. Figure 2.2 shows two objects named A and B interact via a binding object.

Figure 2.2: *RM-ODP binding object.*

Interactions between computational objects are asynchronous and can take three forms. Each form has its own type of interface, but they are collectively called *computational interfaces*.

1. **Operational interfaces.**
   Operational interfaces provide a client-server model for distributed

computing - client objects invoke operations at the interfaces of server objects (i.e. the RPC paradigm). There are two types of operations: *interrogations*, which return a termination and *announcements*, which do not return a termination.

2. **Stream and flow interfaces.**
   Flow interfaces have been included in RM-ODP to cater for multimedia and telecommunications applications which typically handle continuous data types. A flow is characterized by its name and its type, which specify the nature and format of data exchanged [14]. The exact semantics of flows are left undefined in the computational model, thus allowing for the definition of several types of flows, depending of the application domain.

   A stream interface encapsulates a set of flow interfaces which together make up a stream of data. A stream interface signature requires the *causality* (i.e. the direction) of each of the flow interfaces it comprises to be declared. In addition RM-ODP defines the concept of a *complementary* stream interface signature, in which the causality of each flow is reversed [3].

3. **Signal interfaces.**
   Signal interfaces are the lowest level of the communication actions. A *signal* is a pairwise, atomic action resulting in a one-way communication from an *initiating* object to an *accepting* object.

Each interface specification contains an *environment contract* which specifies the QoS requirements imposed by the object on its environment and the QoS provided by the object if it's own requirements are met.

2.3.2 The engineering view

In this, viewpoint a distributed system is modeled in terms of *engineering objects* and *channels*. Engineering objects can be either *basic engineering objects* (corresponding to objects in the computational specification) or *infrastructure objects* (like protocols). Channels correspond to a binding object in the computational specification. Thus, the interaction visualized in Figure 2.2 will be modeled in the engineering language as shown in Figure 2.3. A channel is a compound object as well, but the details are not relevant for

Figure 2.3: *RM-ODP channel object.*

this thesis.

2.4 CORBA

The *Common Object Request Broker Architecture* ***(CORBA)*** is a specification for architectures for distributed computing. It is standardized by the *Open Management Group* ***(OMG)***, which some years ago already numbered more then 800 members.

CORBA is by now well documented in many books and numerous articles. Section 2.4 of [12] lists the following main components of CORBA:

- OMG Interface Definition Language
- Language Mappings
- Operation invocation and dispatch facilities (static and dynamic)
- Object adapters
- Inter-ORB Protocol

Of these we only mention a few details about the *Interface Definition Language* ***(IDL)*** and the object adapters, as they are the only components of direct concern for this thesis.

2.4.1 IDL

The participant entities in a CORBA compliant distributed system relate to each other in terms of being clients or servers (*object implementations*) to one another. During its life-time, an entity can always play the role of a client, or always play the role of a server, or it can play the role of a client toward some entities and the role of a server toward other entities.

In order to be able to invoke operations on a server (an object implementation), a client must know the *interface* offered by the object implementation, that is which *operations* it supports, the type of the operations' arguments and the type of the operations' return values.

In CORBA, such object interfaces are declared by means IDL, which is a high level specification language. The purpose of IDL is to provide programming language independence to CORBA applications. Interfaces declared in IDL must be translated by special IDL compilers to data types and in a regular programming language.

In this thesis, we use ICE, another middleware for distributed computing, which has been influenced also by CORBA in its design. ICE has also an IDL language, which has the same purposes as CORBA's IDL.

As we discuss in Section 5.1, one of the major issues in providing streaming support within a system like CORBA or ICE is the use of the system's IDL.

2.4.2 The CORBA object adapters

In CORBA, see Figure 2.4, an object adapter functions as the glue between the *object implementation* (the *"server"* in the traditional client-server sense) and most of the services offered by the ORB. An object implementation can be implemented, at least in theory, upon different ORBs provided by different vendors.

The object adapter works like a middleware within the middleware, and masks the differences between the different ORBs upon which an object implementation can function. Different ORBs will invariantly provide different levels of certain services and they might offer specific services available from no other ORBs. The environments in which the different ORBs run will also invariably offer varying degrees of properties which the object implementations need in order to serve the clients. Thus, the object adapters existence is justified, as they are that piece of middleware which makes sure the object implementations can run on any ORB.

Quoting from [7]: *"With Object Adapters, it is possible for an Object Implementation to have access to a service whether or not it is implemented in the ORB Core – if the ORB Core provides it, the adapter simply provides an interface to it; if not, the adapter must implement it on top of the ORB Core. Every instance of a particular adapter must provide the same interface and service for all the ORBs it is implemented on."*

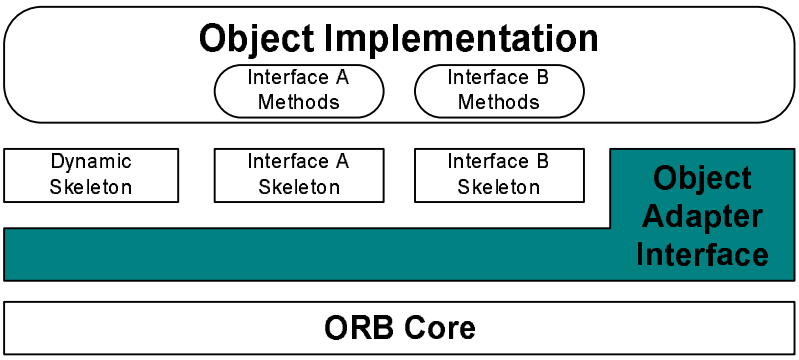

Figure 2.4: *The CORBA object adapter.*

Whereas different object adapters can provide different interfaces and functionality, there is, of course, little point in implementing object adapters which offer mostly the same services and interfaces to object implementations. Indeed, the CORBA specification advises that *"... it is desirable that there be as few [object adapters] as practical. Most object adapters are designed to cover a range of object implementations, so only when an implementation requires radically different services or interfaces should a new object adapter be considered."* [7].

Therefore, the CORBA framework requires only one object adapter to be implemented, the so called *Portable Object Adapter*, or POA for short.

To begin with, CORBA required another object adapter to be implemented, the now very obsolete *Basic Object Adapter*, (BOA). BOA proved to be so poorly specified that implementors had to choose ad hoc solutions for common actions performed by the object adapter, which led to incompatibilities across ORBs. Eventually, OMG came up with the POA specification, which is meant to be a portable alternative of BOA.

A good and concise treatment of the subject of object adapters, both in general and specifically CORBA object adapters, is [26]. The functionality provided by object adapters is summarized as follows:

- **Request demultiplexing.**
  For each request received by the CORBA core, the object adapter will find the appropriate servant which can execute it.

- **Operation dispatching.**
  Once the appropriate servant has been located and identified, the request is passed to it.

- **Activation and deactivation.**
  object adapter do active CORBA servants, which is a process of initialization, sometimes also called *incarnation* of CORBA objects. When servants are no longer needed, they are deactivated, which is also called *etherealization* of CORBA objects.

- **Generating object references.**
  Servants for CORBA objects need to register with the object adapter. Upon registration, the object adapter generates an object reference for each servant, which includes addressing information on how to reach the object in a distributed system.

One of the MULTE project's suggestions for how to provide support for streaming to MULTE-ORBis to provide a new object adapter specifically designed to handle multimedia data.

2.5 CORBA's A/V Streams Specification

The *Audio/Video Stream Specification* ***(A/V Streams Specification)*** [6], sometimes also referred to as the *Control and Management of Audio/Video Streams* specification, is OMG's attempt provide support for multimedia streaming by means of CORBA ORBs.

This specification has been implemented in several ORBs. TAO-ORB is probably the most well known among these. However, not all ORBs have implemented it, even many years after it's initial release.

The A/V Streams Specification is considered *"underspecified and weak"* in regard to its ability to assure inter-ORBs operability [3]. It is also considered *"useful for basic Internet media-on-demand applications"* but *"unsuitable for the construction of arbitrary distributed media-processing applications"*, same referance. We have met similar evaluations in many other texts.

It is because of its lack of suitability that there still is conducted so much research around the world in regard to adding streaming capabilities to object-oriented middleware.

2.6 Summary

In this chapter we have presented systems of general interest for multimedia distributed computing. CORBA has definitely influenced the field of distributed computing for many years now, and the RM-ODP specification is a useful referential framework. An unsuccessful attempt to provide support for streaming to CORBA has been the A/V Streams Specification.

In the next chapter we will present other systems and tools which are directly relevant to the work presented in this thesis.

Chapter 3

Specific background

In this chapter we give an introduction of those frameworks, systems and tools which are directly relevant for this thesis. For each of them, we present only those features which have some bearing for the work presented in this thesis. The reasons why we have chosen exactly these platforms and tools are given in Section 4.2, where we analyze the requirements for our MSA.

3.1 Da CaPo

The ***Dynamic Configuration of Protocols (Da CaPo)*** [23, 5] is a framework for distributed computing which allows the creation and dynamic (*run-time*) adaptation of application specific networking protocols. We will give here a general presentation of Da CaPo based on the original implementation. In Chapter 6 we present our own implementation.

3.1.1 The motivation for Da CaPo

In the early days of networking, the networks were the decisive factor in determining the throughput achieved by computer systems. Even though both general computer hardware and networking hardware have evolved, the latter has evolved faster, turning the situation around. Already for several years ago it has been recognized that the throughput experienced by high speed networks depends on the end-points' capacity to process network traffic. This phenomenon, sometimes called ***slow-software-fast-transmission***, has been traced down to *"the insufficient processing power of end-systems and the complexity and redundancy of end-system protocols. These protocols tend to be designed inefficiently and tend not to be sufficient embedded into operating systems."* [23]

3.1.2 Da CaPo's philosophy

In the approach taken by Da CaPo, individual tasks performed by the traditional protocol stacks are implemented by small separated units called ***protocol functions***. Examples of such tasks are error control, flow control, encryption and decryption and presentation coding.[1] These protocol functions are then used to build highly customized protocol stacks, called ***protocol graphs***, specificaly build for the particular needs of each individual application. The protocol graphs are constituted in such a way as to take into consideration both the network services offered to the application by the particular networking technology on which the application is run, and the concrete available resources at run-time. These protocol graphs can be altered dynamically at run time, so that they are able to adapt to changing networking conditions.

3.1.3 Da CaPo and the OSI referance model

In the traditional OSI referance model, computer communication is achieved by means of a seven layers protocol stack, where end-to-end communication is present from layer 4 and upward. Da CaPo handles the functionality provided by layers 3 to 7 of the OSI referance model, as shown in Figure 3.1.

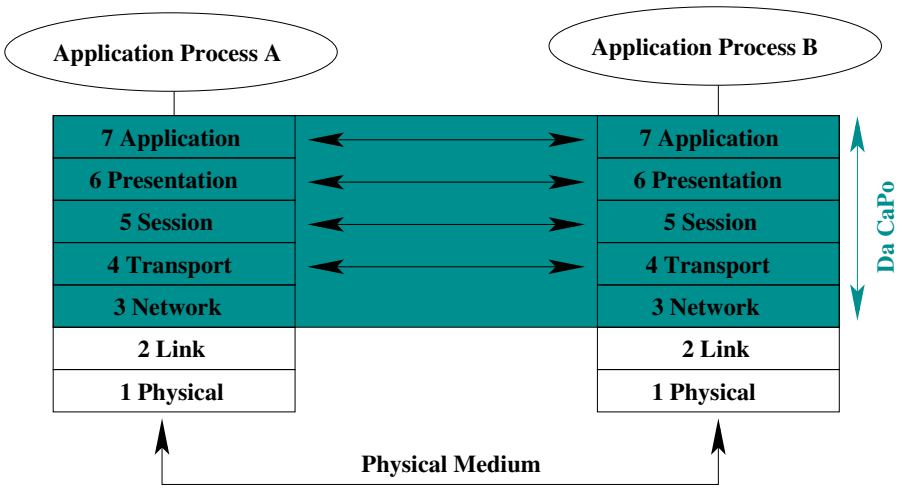

Figure 3.1: *The OSI referance model and Da CaPo*

Da CaPo breaks a complete end-to-end communications system in three parts. These are called the *application layer* ***(layer A)***, the *communications layer* ***(layer C)***, and the *transport layer* ***(layer T)***.

[1] For a more detailed presentation of protocol functions, see the discussion on granularity in Section 4.2.1 of [23].

As shown in Figure 3.2, which has been reproduced from [5], layer C

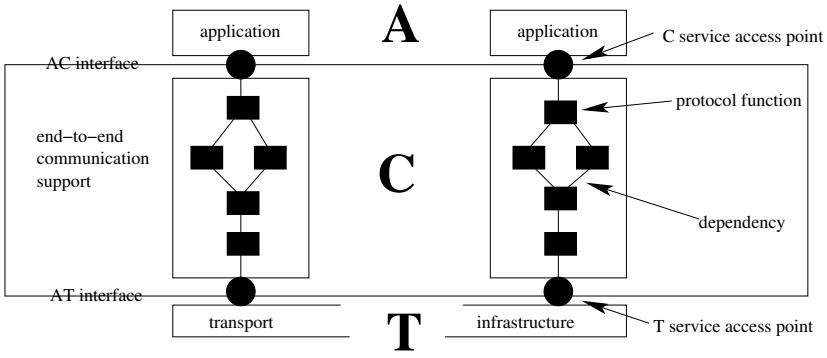

Figure 3.2: *Da CaPo's three layers' model*

covers all aspects of end-to-end communication support (layers 3 to 7 of the OSI referance model).

The application layer and the transport infrastructure (layers 1 and 2 of the OSI referance model) are glued together by Da CaPo's implementation of the communications layer. We can therefore say that Da CaPo consists of the functionality of layer C, the interfaces to layer A and layer T, and a suitable runtime environment.

The purpose of layer C is best described by this quotation from [23] *"In layer C the end-to-end communication support adds functionality to the layer T services in such a way so that at the AC-interface a full set of services is provided to run distributed applications. Layer A corresponds to a set of distributed applications which access the services of the underlying end-to-end communications layer C through the AC-interface. The AC-interface offers functions that permit the application to specify which type of services are needed and quantify the expected QoS."*

The services provided by layer C are put together by a chain of protocol functions, some of them potentially chained in parallell, as it is graphically depicted in Figure 3.2. Data dependencies beetween the protocol functions impose a certain ordering among them. Those protocol functions which datawise are completely independent of each other, if any, are those who can be chained in parallell.

Protocol functions are logical units executing a certain task. They are accomplished by ***protocol mechanisms***[2] and are implemented by means of ***modules***, which are working agents, realized either in software or in hardware.

A protocol function can be implemented by several modules which all provide the same functionality but have different performance characteris-

[2]See Section 4.2 of [23] and Section 2.2 of [5] for more details.

tics. An application can choose which of the available modules it desires to use. As an example, MPG4 decoding can be done both in software and in hardware, with the hardware implementation performing fastest. Modules are always grouped in pairs, consisting of a sending and a receiving module. Because they play such a central role in the Da CaPo architecture, modules are discussed in greater detail in Section 3.1.4.

3.1.4 Da CaPo modules

Modules are the building blocks for any protocol configuration in Da CaPo. Based on their specific purpose, modules can be ***A-modules***, ***C-modules*** or ***T-modules***. C-modules are used to build the very protocol configuration needed by an application. A-modules and T-modules are providing the interface between Da CaPo and the application, and between Da CaPo and the transport infrastructure, respectively; they must not be confused with the application or the transport infrastructure themselves.

In order to allow for unconstrained configuration of protocols, all modules adhere to a well defined interface, called the ***unified module interface***. The unified module interface provides routines which allow modules to be allocated (constructed) and deallocated (destructed), as well as to *get* application data and control information from their runtime environment, and to *deliever* application data and control information to their runtime environment. The unified module interface is not discussed further in this thesis because it is not implemented in the Da CaPo core we present.

Since application data and control information often travel in opposite directions, they are handled separately by the unified module interface. Those modules which do not need to distinguish between application data and control information (modules performing only simple tasks, like computing a check sum) are called ***single modules***, because they have a single data information. Other modules, which must distinguish between application data and control information, (like modules implementing, for example, the *Idle Repeat Request* ***(IRQ)*** function) are called ***double modules*** and have a double data interface. Double modules can be installed in the main path only and are automatically present in both paths of a protocol.

In the original implementation *"modules are handled as* passive *components: They do not actively perform a specific work (like independent processes), but are called by the runtime environment to perform their work (up calls)."* [5]

The modules had to abide to a well defined ***unified module interface***. We do not present the details of the interface here, because we will implement another interface in our own implementation of Da CaPo. We want to point out though, that the A-modules and the T-modules did not have to implement the whole of the unified module interface, because they did not have to interface other modules both *beneath* and *above* themselves in the

protocol graph.

3.2 ICE

The *Internet Communication engine* ***(ICE)*** [10, 9, 8] is a distributed computing platform which might not be so widely known in the ODMP research comunity yet. We give therefore a general presentation of ICE first, before focusing on those features which are of direct relevance for this thesis.

ICE is developed by ZeroC. The reason for creating yet another platform for distributed computing, is that all existing platforms at the time when the initiative to create ICE was taken, had some serious flaws, at least in the eyes of the ICE developers. [8] gives a concise and comparative rationale for ICE.

The design of ICE has been much influenced by the experience gained by the distributed computing comunity during the years prior to the ICE initiative. ICE is often compared to CORBA. This is both because the two of them share many common features, but also because they share the same habitat: they run on the same platforms, they are both free source implementations and they address the same basic needs, as expressed by the RPC pattern of communication. ICE attempts to rectify many of the serious flaws of CORBA, and it especially distantiates itself from CORBA's sometimes unnecessary complexity.

[9] gives a short presentation of ICE, and [10] gives a detailed presentation of ICE, in addition to being an ICE programmer's manual.

3.2.1 ICE in the press

ICE has existed for several years now. At the time of this writting it has reached a level of maturity which makes it appropriate for both research purposes and for the implementation of production quality systems. Recently, ICE has received the attention of two important computer science forums, IEEE and ACM. In February 2004, each of these forums has published an article about ICE in their respective magazines, [9, 8].

In the fall of 2004, ZeroC has made a press release entitled *"Boeing selects Ice for the Future Combat Systems Program"*. This is another proof that ICE is mature enough to attract the attention of large companies which need to implement heavy weight distributed applications.

3.2.2 ICE features

ICE exhibits many of CORBA's features, but not all. It also introduces a few features of its own. Among the features which it has in common with CORBA, there are some which have different semantics. Simply stated, ICE implements all "necessary" features from other platforms, especially from

CORBA. It redefines some of them, like the *object model*, and introduces some new ones, like allowing the declaration of *classes* in its own IDL language, in addition to interfaces.

Figure 3.3 shows the schematic constituency of a client-server application based on ICE. It is strikingly similar to a CORBA based application's structure, as depicted in very many CORBA texts.

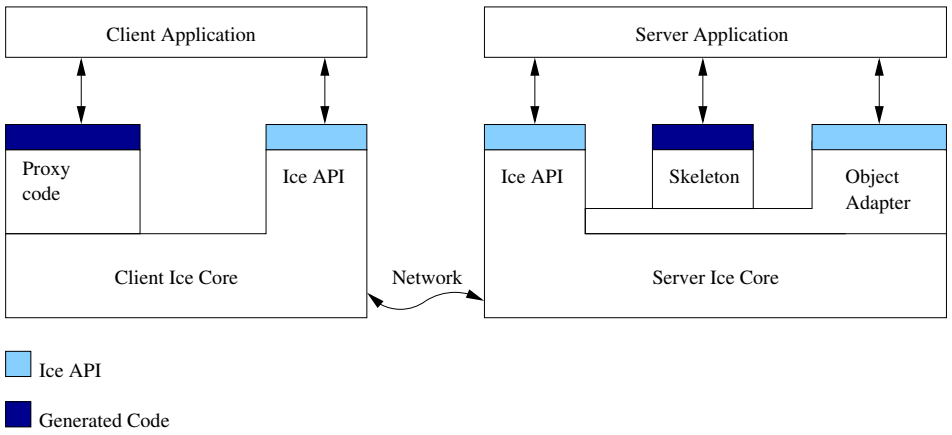

Figure 3.3: *ICE Client and Server Structure, [10]*

The client and the server applications are composed of a mixture of application code, library code and code generated from Slice definitions. We defer the presentation of Slice until Section 3.2.9.

The ICE core, used in both the client and the server applications, provides access to generic services like the initialization of – and the cleaning up after the ICE run time environment.

The proxy code is generated from Slice definitions and is therefore specific for each set of definitions. It provides a *down-call* interface for the client, so that the client can make RPCs by calling functions declared in the proxy code. The proxy code is also providing *marshaling* and *unmarshaling* code.

Also the skeleton code is generated from Slice definitions, and is in the same way as the proxy code, specific to each set of definitions. The skeleton is the server-side "proxy". The skeleton code provides an *up-call* interface which allows the ICE run-time on the server side to transfer a call to application code. Also the skeleton provides the necessary marshaling and unmarshaling code.

The object adapter is a server side specific programing artifact. We dwell more on it in Section 3.2.11.

We present in the reminder of this section a few other features of ICE.

3.2.3 Configuration files' API

ICE provides a rich set of API calls to read and write values from and to configuration files, from within ICE based applications. A typical example of values which should be read from a configuration file, as an application starts up, are the protocol name and the port number to be used for the communication between a client and a server. These values should not be hardcoded, as they are likely to have to be changed from time to time. Alternatively, they can be given to the applications as parameters from the command line.

3.2.4 Supported protocols

With ICE, operations can be invoked by using either TCP, UDP or SSL[3]. Each protocol transmits its properties to the operations which use them. For example, an operation invoked via TCP is guaranteed to be delievered to its receptor, or the sender will be notified of the failure to do so. An operation sent via UDP might be lost without the sender being notified about it, but it will usually be delievered faster then an operation sent via TCP, because it necessitates less processing on the end-point nodes. With SSL, the invocation of operations can be encrypted, in addition of being guaranteed to be delievered. The cost is additional processing time on the end-point nodes.

3.2.5 Invocation styles

ICE supports the following invocation styles:

- **Synchronous Method Invocation**.
  This is the default invocation dispatch mode, and it implements the traditional concept of RPC.

- **Asynchronous Method Invocation (AMI)**.
  In this mode, clients pass an extra parameter to the servers with each invocation. This parameter is a *callback object*, and do not block waiting for an answer from the server. When the answer is available, the servers use the callback object to inform the client about it.

- **Asynchronous Method Dispatch**.
  This mode, is the server side equivalent of AMI. Normally a server processes an invocation at once it receives it. In this mode, it assigns the invocation to an available thread and leaves it to it to process it. In this way the server is free to continue doing other things.

[3]Secure Socket Layer

3.2.6 Invocation types

Depending on the protocol used for invocations, ICE supports the following invocation types:

- **Datagram invocations**.
  Datagram invocations have "best effort" semantics and are sent over UDP connections. A datagram invocation can be made only for operations without returning parameters.

- **Batched datagram invocations**.
  Batched datagram invocations are accumulated in a buffer, before they are sent all together in a single datagram invocation. They can lead to a significant reduction of processing overhead, especially for short messages. The total amount of data of a batched datagram invocation must not exceed the size of the network's PDU, otherwise UDP fragmentation will occur, which greatly increases the chance that the whole batch of invocations must be discarded, if even only one fragment is lost.

- **Oneway invocations.**
  Oneway invocations of operations have "best effort" semantics, but they are sent over a connection oriented protocol, like TCP or SSL. This assures ordered delievery of invocations to the receiver, but the threading policy of the server might lead to the invocations being processed in another order.

- **Batched oneway invocations.**
  Batched oneway invocations are buffered on the client side of the application and are sent in a single message to the server. They can lead to a significant reduction of processing overhead, especially for short messages. Batched oneway invocations are server by a single thread in the server, so ICE guarantees that they will be served in the order of delievery.

3.2.7 ICE services

ICE implements a number of *services* often needed by applications. We only mention them in this thesis, as we do not make use of any of them. They are:

- **IcePack** - a location service.

- **IceBox** - a component managing service.

- **IceStorm** - a distribution switch for events.

- **IcePatch** - a software patching service.
- **Glacier** - a firewall service.

3.2.8 QoS in ICE

ICE supports a very limited notion of QoS. In connection with its IceStorm service, ICE supports *"only one QoS parameter"*, called reliability [10]. The possible values are oneway and batch, with oneway as default. This property affects *how soon* messages are delievered, as explained in Section 3.2.5.

3.2.9 Slice

Slice is ICE's equivalent of CORBA's IDL. A thorough presentation of Slice is given in Chapter 4 of [10]. We present here only those features which we use in our work.

Classes

Slice supports the declaration of *classes* in addition to *interfaces* which is common for an IDL, like CORBA IDL.

Slice classes can have operations, like interfaces, and data members, like structures. This leads to hybrid objects which can be passed both by reference, as interfaces are, or by value, as any variables. The feature we use in our implementation is that *"classes allow behavior to be implemented on the client side, whereas interfaces allow behavior to be implemented only on the server side"* [10].

Slice classes exhibit the most common features of classes in object-oriented programming languages. They can be *inherited*, with single inheritance only, and can be self referential. In addition, they can be used to implement interfaces.

The byte data type

byte is one of the 8 data types supported by Slice. byte *"is an (at least) 8-bit type that is guaranteed not to undergo any changes in representation as it is transmitted between address spaces. This guarantee permits exchange of binary data such that it is not tampered with in transit. All other Slice types are subject to changes in representation during transmission"* [10].

We use this data type in the demos of this thesis, to assure that the images we stream have the same representation on the receiver as they had on the sender.

Slice to C++ language mappings

All features of Slice have a language mapping to C++[4]. The language mapping is accomplished by ICE's Slice to C++ compiler, called slice2cpp.

There are a few features of the Slice to C++ mapping which we need to mention. Among these are the C++ types which are declared from Slice classes.

In the reminder of this sub-section, we freely quote sentences from Section 6.14 and 6.11 of [10], without using quotation marks.

Slice classes are mapped to C++ classes with the same name. For each data member of a Slice class, the generated class will contain a public variable. For each operation of a Slice class, the generated class will declare a virtual function. Consider the following Slice class declaration:

```
class TimeOfDay {
  short hour;         // 0 - 23
  short minute;       // 0 - 59
  short second;       // 0 - 59

  string format();  // Return time as hh:mm:ss
};
```

For this class, the following (simplified) C++ mapping will be made:

```
class TimeOfDay : virtual public Ice::Object {
public:
  Ice::Short hour;
  Ice::Short minute;
  Ice::Short second;

  virtual std::string format() = 0;

  ...
};

typedef IceInternal::Handle<TimeOfDay>
    TimeOfDayPtr;
```

Notice the typedef made after the class declaration. This type implements a *smart pointer* that wraps dynamically-allocated instances of the class. In general the name of this type is <class-name>Ptr.

The TimeOfDay C++ class is the *skeleton* class, see Figure 3.3, for this Slice definition. Object implementations for the TimeOfDay Slice declaration are implemented in a server by inheriting the TimeOfDay C++ class.

If we let TimeOfDayI be the class which inherits TimeOfDay for the purpose of implementing TimeOfDay objects in a server, we assign a new instance

[4]ICE also provides a language mapping to Java, and native support for PHP. At the time of this writing, release 1.6 of ICE is expected, which is supposed to introduce a language mapping for Python 2.3 as well.

of TimeOfDayI to a smart pointer variable, in order to let ICE's run-time take care of all memory management:

```
{
  TimeOfDayPtr tod = new TimeOfDayI;
  tod->second = 0;
  tod->minute = 0;
  tod->hour = 0;
  ...
}
```

tod is now our object implementation in a server.

In addition to this mapping, a class declaration also shares in the language mapping typical for ICE interfaces: a Slice class will also be mapped to a *proxy handle* type. For the same Slice declaration as above, the compiler will generate also the following (simplified) C++ definitions:

```
namespace IceProxy {
  class TimeOfDay : public virtual Ice::Proxy::Ice
      ::Object {
    ...
  };

  typedef IceInternal::ProxyHandle<IceProxy::
      TimeOfDay> TimeOfDayPrx;
}
```

The general name for a proxy handle is <class-name>Prx. In the client's address space, an instance of IceProxy::TimeOfDay is the local ambassador for a remote instance of a TimeOfDay object implementation in a server. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance. This type corresponds to the proxy depicted in Figure 3.3.

Invoking the format operation on a TimeOfDayPrx instance in the client will be dispatched by ICE's run-time to an object implementation, like tod, in the server.

3.2.10 The ICE development cycle

The development cycle of an ICE application, depicted in Figure 3.4, resembles very much the programming cycle of a CORBA application.

We note in this figure, the 3 *human* sources of input, depicted with ovals: the Slice developer. the client developer and the server developer. These 3 can be the same person or not. The compiler developed within the work with this thesis will play all these 3 roles, as far as the ICE development cycle is concerned. A human programmer must still provide a FIDL++ specification, and he or she may still be involved in developing other functionality, such as a GUI, but as far as ICE programing is concerned, our compiler will generate all the code that is necessary.

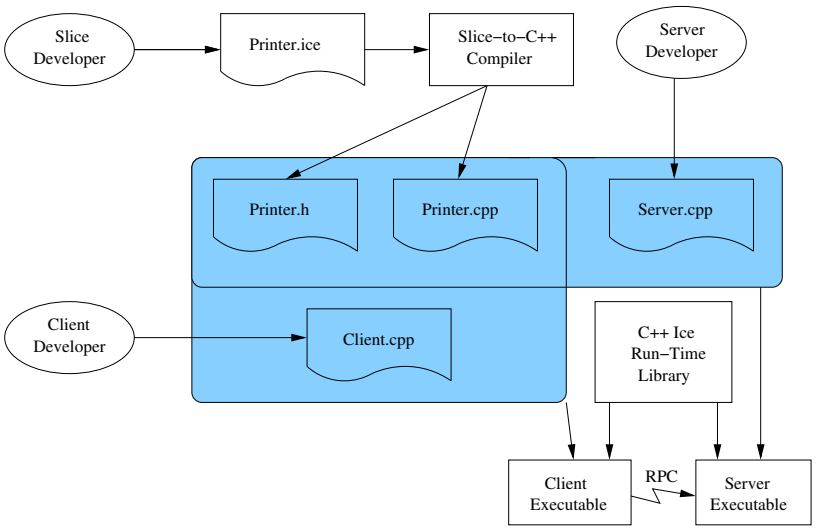

Figure 3.4: *ICE development cycle, [10]*

3.2.11 The ICE object adapters

In ICE, the concept of the *object adapter* is closely related to the concept of a *communicator*. The communicator is the main entry point to the Ice run-time on the server side, and it is entirely declared in Slice.

A communicator will be associated with a number of run-time resources. Among these it will contain one or more object adapters. The object adapter is also entirely declared in Slice.

Shortly stated, the purpose of the ICE object adapter is to *"dispatch incoming requests and take care of passing each request to the correct servant"* [11]. We also reproduce the following list of responsabilities of the object adapter from [11]:

- It maps Ice objects to servants for incoming requests and dispatches the requests to the application code in each servant (that is, an object adapter implements an up-call interface that connects the Ice run time and the application code in the server).
- It assists in life cycle operations so Ice objects and servants can be created and existing destroyed without race conditions.
- It provides one or more transport endpoints. Clients access the Ice objects provided by the adapter via those endpoints. (It is also possible to create an object adapter without endpoints. In this case the adapter is used for bidirectional callbacks...

As we can see, the purpose of ICE's object adapter, is very closely related to the purpose of the object adapter in a CORBA implementation.

3.3 FIDL

The Flow Interface Definition Language ***(FIDL)*** [16] is a specification language which allows the programmer to describe the characteristics of flows of multimedia data.

FIDL's flows specifications resemble the concept of flow interfaces introduced by the computational view of RM-ODP, even though [16] does not mention RM-ODP at all. We will therefore refer to *flow specifications* in this thesis rather then to flow interfaces.

The language has been used in [16] to algorithmically compare flows specifications to each other in order to find common sets of flow characteristics. The result of such a comparison is an *intersection* of flow specifications. Among other things, the intersection of flow specifications is an accurate description of end-points compatibility. In the following sections we present the main features of FIDL and give an example of a FIDL specification.

3.3.1 Media format taxonomy

In FIDL, a *stream* is made of one or more *flows*, and each flow contains one or more declarations of multimedia *elements*. A *constraint* clause is then used to specify which combinations of flow elements can constitute legal flows in a stream.

The multimedia element types provided by FIDL have been divided into several categories, or *classes*, which form the hierarchy shown in the following figure, which has been reproduced from [16] :

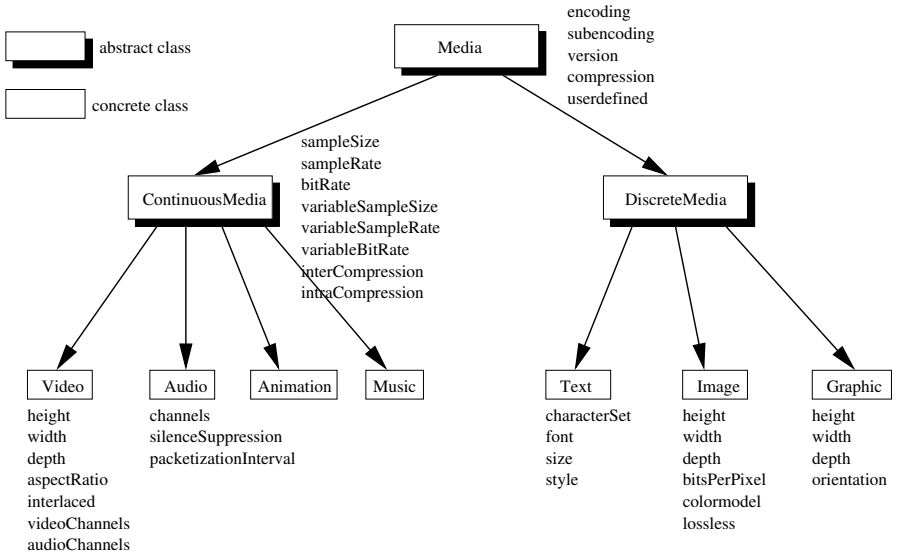

Figure 3.5: *FIDL element class hierarchy [16].*

Generic element types	Other keywords
animation	constraint
audio	flow
graphic	sink
image	source
music	stream
text	
video	

Table 3.1: FIDL keywords: generic keywords [16].

Keywords (attribute names)			
aspectratio	depth	orientation	userdefined
audiochannels	encoding	packetizationinterval	variablebitrate
bitrate	font	samplerate	variablesamplerate
bitsperpixel	height	samplesize	variablesamplesize
channels	interlaced	silencesuppression	version
characterset	intercompression	size	videochannels
colormodel	intracompression	style	width
compression	lossless	subencoding	

Table 3.2: FIDL keywords: attribute names [16].

As we can see, on each level, a class of elements has an associated list of attributes, specific to that class. Sometimes the attributes repeat themselves, as they are specific to several classes, like the *height, width* and *depth* attributes which can be found in both the Video, Image and the Graphic classes. We note however, that these classes are on the same level in the hierarchy. By inspection we can see that the attributes do not repeat themselves on several levels of the class hierarchy, but are *inherited* from a base class to a derived class, as in regular object-oriented programming.

We do not know if the attribute-belongs-to-a-class or the attribute-belongs-to-a-class-level design has been enforced in the implementation of [16], but we will came back to this issue when we present the interpretation we have given to the FIDL language.

3.3.2 Lexical conventions

The keywords of the FIDL language consist of all the attribute names presented in Figure 3.5, a set of *generic element types* and 5 *other keywords*. Tables 3.1 and 3.2 are reproduced from [16] and they show all the keywords of FIDL:

3.3.3 FIDL grammar

The grammar of FIDL is specified in [16] in a notation similar to the *Extended Backus-Naur Form* ***(EBNF)***. We reproduce it here:

```
<stream_spec>       ::= "stream" <identifier> "{" <
    flow_spec>+ "}"
<flow_spec>         ::= <direction> "flow" <identifier>
                        "{" <element_spec>+ [<
                            constraint>] "}"";"
<direction>         ::= "sink" | "source"
<element_spec>      ::= <generic_name><label>
                        ["{" <attribute>+ "}"] ";"
<attribute>         ::= <attr_name> "=" <attr_value>
                        ("+" <attr_value>+)* ";"
<attr_value>        ::= <atomic_value>
                      | "{" <set_value> "}"
                      | "(" <range_value> ")"
<set_value>         ::= <atomic_value> ("," <
    atomic_value>)*
<range_value>       ::= <atomic_value> "," <
    atomic_value>
<atomic_value>      ::= <string>
                      | <integer>
                      | <float>
<constraint>        ::= "constraint" <constraint_expr
    > ";"
<constraint_expr>   ::= <or_expr> ("|" <or_expr>)*
<or_expr>           ::= <and_expr> ("&" <and_expr>)*
<and_expr>          ::= <label>
                      | "(" <constraint_expr> ")"
<label>             ::= <identifier>
```

3.3.4 A semantic interpretation of FIDL

We take now a closer look at the possible semantic meanings of FIDL's grammar. If some things seem a little difficult to understand at the first reading, we believe they will become very clear once we come to an example of a FIDL specification, such as the one given in Section 3.3.5.

By analyzing the grammar presented above, and the examples presented by the authors of FIDL, we see that a FIDL specification is actually the specification of a ***stream*** of data of one or multiple types.

A stream specification contains one or more ***directed flow*** specifications. The general syntax of a stream specification is therefore:

```
stream STREAM_NAME {
  <directed_flow_specifications>
}
```

In this context, the capitalized label "STREAM_NAME" represents a generic identifier. The meta-code in triangular parenthesis "<directed_flow_specifications>" suggests that the body of a stream specification consists of a list of directed flow specifications, and that this is not real FIDL code. We will use this notation in the reminder of this section to suggest where identifiers are expected and what is meta-code. All other symbols are keywords or required punctuation.

A flow's direction is specified by the source or sink keywords. Each flow specification contains one or more ***element*** specifications and a single, *optional* ***constraint*** clause. The general blue print of a directed flow specification is:

```
<direction> flow FLOW_NAME {
  <element_specifications>
  <constraint_clause>
};
```

An element specification has the following general form:

```
<element_type> ELEMENT_NAME {
  <attribute_specifications>
};
```

The (list of) attribute specifications of an element specification can be empty. It is not clear to us why FIDL allows this scenario. One conceivable use would be to allow the declaration of element *variables* inside a flow specification, but none of the examples presented in [16] makes use of this syntactic liberty. Neither CORBA IDL nor Slice allow the declaration of stand alone variables.

If the list of attribute specifications is not empty, it consists of one or more assignments. An assignment assigns an *appropriate value* to an attribute name. An appropriate value can be *atomic*, which means it is a single value of any of the legal data types of FIDL, or it can be a *set* or a *range* value. The legal data types of FIDL are integer, float or string.

It is interesting that a set or a range value is allowed to have elements of different types.

While we like very much the ability to specify sets and ranges in FIDL, a strict interpretation must be given to their semantic meanings, in an implementation.

The constraint clause of a flow, if present at all, specifies which of the declared elements must be present in an implementation in order to have a legal and complete flow. For this purpose the *or* "|" and the *and* "&" operators are defined. If the contraint clause is missing, but the list of element specifications is not empty, [16] specifies that any of the declared elements can be part of the flow at run time, but they are not required to.

3.3.5 A FIDL specification example

Several FIDL specifications are given as examples in [16]. These examples are used in [16] to illustrate different sides of the system implemented therein, like communicating with a database for multimedia objects. We do not reproduce here any of those example, as we want to focus in this thesis on an application of FIDL specifications which has only peripherally been mentioned in [16]. Our goal is to present a solution to the challenging problem of handling the concept of streams of multimedia data as first class objects in the context of middleware for distributed computing.

As a demo, we have implemented a simple *one-way* video telephony application. We make use of common hardware, like a webcam, a microphone and the host computer's soundcard and screen.

Following the guidelines presented in the previous section, we declare first a stream which we choose to call "WebCamChat":

```
stream WebCamChat {
  ...
}
```

Then, we declare a source and a sink flow. The source flow represents the end-point where the webcam and the microphone reside, and the sink flow specification represents the end-point where the images and the sound are consumed. We give the flows different names in order to be able to distinguish them without referring to the their direction:

```
source flow WebCam1 {
  ...
  constraint ...
};

sink flow WebCam2 {
  ...
  constraint ...
};
```

Because of the limitations of the hardware used, we have chosen to declare 3 different image elements and an audio element for each of the flows. We show here the elements for the source flow. (The audio element of the sink flow specification will be named something else then Mic. Otherwise the two flow specifications are equal.) The elements have the following declaration:

```
image QSIF {
  encoding = "YUVp420";
  samplerate = (5, 30);
  height = 120;
  width = 160;
};
```

```
image SIF {
  encoding = "YUVp420";
  samplerate = (5, 30);
  height = 240;
  width = 320;
};

image VGA {
  encoding = "YUVp420";
  samplerate = (5, 15);
  height = 480;
  width = 640;
};

audio Mic {
  encoding = 2;
  channels = {1, 2};
  samplerate = {8000, 11025, 22050, 44100};
  samplesize = {8, 16};
};
```

What we want our application to provide is a combination of any of the image elements and the audio element for each flow. We provide therefore the following constraint to each of the flow specifications:

```
// constraint for the source flow
constraint (VGA | SIF | QSIF) & Mic;

// constraint for the sink flow
constraint (VGA | SIF | QSIF) & SoundCard;
```

Putting it all together, here is our demo's specification code:

Listing 3.1: FIDL specification for our demo application.

```
1  stream WebCamChat {
2    source flow WebCam1 {
3      image QSIF {
4        encoding = "YUVp420";
5        samplerate = (5, 30);
6        height = 120;
7        width = 160;
8      };
9
10     image SIF {
11       encoding = "YUVp420";
12       samplerate = (5, 30);
13       height = 240;
14       width = 320;
15     };
16
```

```
17      image VGA {
18        encoding = "YUVp420";
19        samplerate = (5, 15);
20        height = 480;
21        width = 640;
22      };
23
24      audio Mic {
25        encoding = 2;
26        channels = {1, 2};
27        samplerate = {8000, 11025, 22050, 44100};
28        samplesize = {8, 16};
29      };
30
31      constraint (VGA | SIF | QSIF) & Mic;
32    };
33
34    sink flow Webcam2 {
35      image QSIF {
36        encoding = "YUVp420";
37        height = 120;
38        width = 160;
39      };
40
41      image SIF {
42        encoding = "YUVp420";
43        height = 240;
44        width = 320;
45      };
46
47      image VGA {
48        encoding = "YUVp420";
49        height = 480;
50        width = 640;
51      };
52
53      audio SoundCard {
54        encoding = 2;
55        channels = {1, 2};
56        samplerate = {8000, 11025, 22050, 44100};
57        samplesize = {8, 16};
58      };
59
60      constraint (VGA | SIF | QSIF) & SoundCard;
61    };
62  }
```

As we will see in the next subsection, there are situations when two flow specifications such as the ones used in our demo should actually be

placed in separate files, as they would each reside on a different machine. The source and the sink end-points do not necessarily have to know about each-other's streaming capabilities, as other entities will care to handle them together. The grammar of FIDL allows though a stream to have several flow specifications. This could be to allow the processing of collocated scenarios. We consider though that collocated scenarios do not necessarily have to be specified in the same file, because two files can just as easily be processed on the same node.

It is also unclear to us why the FIDL grammar allows the declaration of more than two flows within the same stream specification. We note also that it is also legal to declare several flows with the same direction within the same stream specification. Maybe the intent of this liberty is to be able to programmatically chose between one of the flow specifications at compile time.

3.3.6 Other features

The Intersector

In addition to its ability to specify streams, [16] presents many additional features of the system of which the FIDL language is part of. Among these features we want to particularly mention the Intersector. Quoting from [16], *"the Intersector is responsible for computing the intersection of the quality and structural interpretations of a pair of flow types."* In other words, given two FIDL files, the intersector will analyze the structure of each stream specification and determine wheather the two specification are compatible at all. If they are, the intersector will also compute the set of common elements of each stream and the set of common attributes with their common attribute values for the two streams.

From our point of view this is an important feature of this system and we will use it as a starting point for the solution we present in this thesis.

[16] gives at least a couple of examples of computations of stream specification intersections. We do not have their system compiled and operative, but we will present here what we understand would be the result of intersecting the two flows specified for our demo application. Before we do that, we want to mention a couple of other things. We quote again from [16]: *"The Intersector is also able to produce the equivalent FIDL code for any stream or flow object. This is useful, for instance, when ... analyzing the result of an intersection."*

Looking at the Program Listing 7.2 on page 59 in [16], we see that the Intersector outputs its results in a syntax which seems to be a subset of FIDL's syntax. By this, we refer to the fact that the values on the right sides of the assignments are all converted to sets, which is a reasonable thing to do in regard to the further processing of this output. These sets can

be very long[5], but since they are machine processed we should not be so concerned about them. Since our sets contain many elements we use "..." in the next listing to indicate that we have not listed all values. The Intersector will of course have to provide a complete listing.

We also notice that since this is an *intersection* of two stream specifications, the flows' directions are now specified as void and that the Intersector provides machine generated names instead of the identifiers given by the authors of the FIDL files to the stream and the flows.

Here is what we think would be the result of intersecting our two flow declarations:

Listing 3.2: Intersector output in FIDL format for our demo's specification.

```
1  stream strIntersection {
2    void flow flwIntersection0 {
3      image lab0 {
4        encoding = {"YUVp420"};
5        samplerate = {5, 6, 7, ..., 28, 29, 30};
6        height = {120};
7        width = {160};
8      };
9
10     image lab1 {
11       encoding = {"YUVp420"};
12       samplerate = {5, 6, 7, ..., 28, 29, 30};
13       height = {240}
14       width = {320};
15     };
16
17     image lab2 {
18       encoding = {"YUVp420"};
19       samplerate = {5, 6, 7, ..., 13, 14, 15};
20       height = {480};
21       width = {640};
22     };
23
24     audio lab3 {
25       encoding = {2};
26       channels = {1, 2};
27       samplerate = {8000, 11025, 22050, 44100};
28       samplesize = {8, 16};
29     };
30
31     constraint (lab2 | lab1 | lab0) & lab3;
32   };
33 }
```

[5]Long from the human point of view. A set with even as many as 10.000 integer values, while large, is totally processable by todays computers.

"Plussed" values

One last remark about FIDL. One thing which can be seen from a careful reading of the FIDL grammar is that we can have *plussed* values on the right side of assignments in element declarations. That is, we could specify an assignment like

```
samplerate = (10, 15) + 7 + 18 + (25, 30);
```

We note the expressive power of this assignment: the right side is made of a *sum* of atomic and range values. Nothing hinders us to also sum sets to atomic values and ranges. We point out though that this will not add anything to the expressive power of the FIDL assignment statement, because the only operator allowed on the right side of an assignment is *plus* "+". The middle values in the assignment above, 7 and 18, can also be viewed as the members of a set, and we could write the above assignment like this:

```
samplerate = (10, 15) + {7, 18} + (25, 30);
```

We note also that the values given (integers in this case) are not in ascending order. Such an assignment must therefore be interpreted somehow.

3.4 Summary

In this chapter we have provided an overview of the main features of those systems and tools which we use for the implementation of our work.

In the next chapter we present the requirements posed to our work, and provide also an analysis of them.

Chapter 4

Requirements

In this chapter, we present the requirements for the MSA. We present first on overview of the requirements in Section 4.1 and proceed with an analysis of the requirements in Section 4.2. The purpose of this analysis is to derive which platforms, systems and tools are the most appropriate for the work we want to accomplish in this thesis.

4.1 Overview

The requirements posed to our MSA have been influenced by the fact that the work presented in this thesis belonged to begin with to the MULTE project and has been continued then on its own. It is therefore natural that we first present the requirements derived from the MULTE-ORB. Then we present the requirements to begin with, and that for the MSA in three stages: 1) MULTE-ORB requirements, 2) Da CaPo requirements and 3) object adapter requirements.

4.1.1 MULTE-ORB requirements

In [21], the following six requirements have been defined for the MULTE-ORB:

1. **Dynamic QoS support.**
   Applications should be able to specify QoS requirements and to change them dynamically. The middleware (MULTE-ORB) should provide the requested QoS and adapt to changes in QoS requirements, resource availability, etc.

2. **Evolution of QoS requirements.**
   New media types and new applications might introduce new QoS characteristics. In order to support these new requirements, QoS management in the middleware must be extensible.

3. **Transparency versus fine grained control.**
   Developers of application components and users should be able to define QoS requirements in high-level (application) terminology, e.g. good video quality, and in low-level system parameters to directly influence middleware configuration, e.g. compression = MPEG, and resource allocation, e.g. throughput $\geq$ 1 MB/s.

4. **Policy control.**
   The middleware should enable end users, application developers, and system managers to specify policies for QoS mapping, negotiation, monitoring, adaptation, etc. For example a policy might be used to express that in case the quality of a video should be degraded, adaptation is done by reducing the frame rate instead of the resolution.

5. **Automatic support for compatibility control.**
   The middleware should detect incompatibilities in user requirements and equipment and resolve them - if possible - with media scaling and transcoding.

6. **Support for seamless system evolution.**
   The integration of new components in the middleware should not require recompilation or changes of existing components and middleware entities. This must also be true for components that encapsulate resources, i.e., APIs to network services.

These requirements were given for this thesis since the MSA was originally intended for the MULTE-ORB.

4.1.2 Da CaPo requirements

The general guidelines for an implementation of Da CaPo have been established at the beginning of this thesis, in cooperation with other MULTE project members. Therefore, we provide in this thesis only a possible implementation of a Da CaPo core, base on these guidelines. The main requirements posed to Da CaPo are:

1. **Use C++.**
   Other project members were already working at a new Da CaPo implementation when the work with this thesis was started. They used C++ for it, because the rest of the system they worked on was to be implemented in C++. When their work with Da CaPo stagnated temporarily, we considered it natural to try to provide an implementation of a Da CaPo core in the same language, so that as much as possible of our work could later be incorporated into the new Da CaPo.

2. **Simple API**.
   Da CaPo modules must adhere to a well defined interface in order for them to be treatable as modules. A module API is a reflection of this interface. It is always a benefit to keep an API as simple and as logical as possible.

3. **Flexibility**.
   The original version of Da CaPo was very flexible, in the sense that it allowed the creation of application specific protocols by means of having each module implement a protocol function of very fine granularity, i.e. each module performed only a simple networking operation. We want to preserve this quality in the new Da CaPo implementation. We note that it does not hinder us from implementing *heavy* modules, which perform complex sequences of networking operations, if we need to.

4. **Adaptability**.
   Applications based on the original Da CaPo system could reconfigure their protocol graphs at run-time, if external factors, such as a drop in the available bandwith, required it. We want to preserve this feature in the new Da CaPo implementation as well.

Also these requirements were more or less given to this author, as we have no intention in this thesis to depart from the work which others are trying to accomplish in the MULTE project. On the contrary, we consider it a benefit to remain as compatible as possible with the project. The fact that this thesis has evolved on its own much of the time is due to external working assignments and not to disagreement on technical grounds.

4.1.3 Object adapter requirements

An object adapter is one of the most important components of a middleware platform like CORBA or ICE. Since the regular object adapters provided by these platforms do not support streaming of data, we have an additional requirement, that our MSA shall extend the capabilities of an object adapter with support for streaming.

4.2 Analysis

In this section, we analyze the requirements posed to our work, in order to decide the most appropriate platforms and tools need for its implementation.

4.2.1 MULTE-ORB requirements revisited

The requirements identified for MULTE-ORB, presented in Section 4.1.1, are general, in the sense that any platform with support for multimedia

streaming must support the *principle* behind each of them. They are of course adapted to the specifics of MULTE-ORB, but they reflect general needs. Since this thesis is not part of the MULTE project anymore, we can not derive explicit consequences for our design and implementation from them. However, they too can contribute to the choice of tools and systems used in our implementation, by providing guidelines. The six requirements were:

1. Dynamic QoS support.
2. Evolution of QoS requirements.
3. Transparency versus fine grained control.
4. Policy control.
5. Automatic support for compatibility control.
6. Support for seamless system evolution.

The first three requirements deal directly with the need for QoS management. They have been stated because, at the time of this writing, there are no object-oriented middleware platforms today which provides support the specification and management of QoS. The closest we come to "native" QoS support in an CORBA ORB is the QoS features of A/V Streams Specification, which has been presented in Section 2.5. Also, ICE has virtually no support for QoS management, as mentioned in Section 3.2.8. The important question for this discussion is which existing platforms or tools are most suitable for the implementations of object-oriented middleware which will satisfy these requirements? Many texts, including [22, 15, 23, 1], suggest that in order to achieve QoS regulated behavior of multimedia applications, support for QoS is necessary at all levels of a system, including the operating system. This is why, to begin with, the MULTE project wanted to use Chorus, and later Real-Time Linux, for their implementation. On the same token, Sumo-ORB was built on a Sumo-ORB core which consisted of more than 20.000 lines of C code [1], which provided the low level implementation of the functionality necessary to manage QoS on higher levels.

In this thesis, we have chosen not to look into the details of how to build a system on the API provided by an operating system like Real-Time Linux. Referring to our discussion of QoS in Section 2.1.4, we concentrate on QoS at the application level.

In light of the discusion of Section 5.1, we consider that application level QoS parameters can be specified both by extending an IDL language, or by means of another, special purpose, language such as FIDL. The extensions to CORBA IDL proposed in [1] are not enough to allow the declaration of application level QoS parameters, because their extensions do not allow

the assignment of values to names, i.e. one can not specify something like framerate = 20 in the extended IDL[1].

For this thesis, we prefer to use FIDL, which is specifically designed to facilitate the declarations of streams and flows of data, including the associated QoS requirements, rather than extend an IDL language even beyond the proposals presented in [1].

In regard to requirement number 4 above, it is beyond the scope of this thesis to concern ourselves with mechanisms for specification of policies. We are more concerned with providing a streaming API which can be used by the application to adapt itself to the employed policy, regardless of how it is specified. Within the MULTE project, however, the issue of policy specification is addressed for instance in [24].

For the fulfillment of requirement number 5, we consider that the FIDL specification language, together with the work presented in [16] (a FIDL specifications Intersector) or [24] (a FIDL based Gateway Trader) are obvious best candidates. Both of these two works implement algorithms for flow interface compatibility control. Of the two, the work presented in [16] has been implemented first, and the work in [24] builds on it. In this thesis, we also build on [16] (FIDL), and we assume the existence of a system, like the above mentioned Intersector, which can provide us with the result of flow specifications compatibility checks. Alternatively to such a system, a programmer can provide our compiler with the result of the intersection of flow specifications for which he or she desires to implement a MSA. As we will see in the reminder of this thesis, this is easily accomplished, and it is the approach we have chosen for our demo application.

The implementation of requirement number 6 provides a distributed system for multimedia with much elegance and user friendliness. Much of this functionality was planned to be implemented in the MULTE-ORB by integrating the Da CaPo system into the ORB. In this thesis, we provide ourselves an implementation of a Da CaPo core which can be used as the starting point for a Da CaPo implementation which exhibits the qualities highlighted in requirement no 6. This requirement is very much alike requirement number 4 of Section 4.1.2.

4.2.2 Da CaPo requirements revisited

The requirements posed to our Da CaPo core implementation are not of a nature to influence the choice of platforms and tools.

In the earlier stages of the MULTE project, the COOL ORB has been considered to be an environment more suitable than other ORBs to host a system like Da CaPo. However, the idea of using COOL has been discarded and this author's literature studies have not led to the discovery of

[1] Just as it can not be specified in CORBA IDL or Slice

a framework for distributed computing which is especially well adapted for Da CaPo.

We believe though, that the use of an operating system with built in QoS management abilities, like Real-Time Linux, would be beneficial. However, this would not be beneficial for Da CaPo alone, but for any system which addresses the issues of QoS management. As we have mentioned in the previous section, we do not try to interface such a special purpose operating system in this thesis.

4.2.3 Object adapter requirements revisited

The original idea was to extend the capabilities of a CORBA ORB's object adapter with streaming capabilities, within the work with the MULTE-ORB. After COOL has been abandoned, other CORBA ORBs have been considered, like TAO-ORB and OmniORB. These ORBs have been considered, because at that time, they were the most mature, most rich in implemented features and they had the most actively developing communities. We have restricted our studies to open source platforms, because they were most readily available, and they are the only ones which allow us to gain in depth insight about their inner workings.

While studying OmniORB, our attention has been directed to the ICE platform, by a posting on a newsgroup. Reading ICE's documentation quickly convinced us that it would be advantageous for us to use it instead of a CORBA ORB.

The feature which attracted us most was the obvious simplicity of ICE's object model. While much simpler then CORBA's, it is still very powerful. This simplicity makes ICE's learning curve much leaner, so that it requires less time and effort to become productive with ICE than with CORBA.

Eventually, we discovered several other benefits of using ICE as the development platform in this thesis, such as the fact that Slice has built in the notion of classes. This allows the implementation of object functionality on the client side of an application too. The functionality of an interface is always implemented only on the server side, both in CORBA and in ICE. However, by the time we discovered these benefits, we had already decided to use ICE rather than a CORBA ORB.

At the time of this writting, version 1.5.0 of ICE has been released, and according to an announcement made by ZeroC in september 2004, release 1.6.0 is due in the 4th quarter of 2004. In this thesis, we use version 1.4.0, because it was the latest release at the time when we began to implementations of our MSA. However, due to our design choices, we believe that our MSA implementation will work with all releases of ICE in the forseeable future.

4.3 Summary

In this chapter, we have presented the requirements for the work of this thesis. Some of these requirements, such as the ones for the MULTE-ORB, have been defined prior to the commencement of our work. Some of the others, like the ones for Da CaPo, have been developed in cooperation with other MULTE project members. Also the requirement to extend an object adapter with streaming capabilities has been given from the very beginning, but only in general terms. *How* to extend an object adapter is the realm of this thesis.

The following chapters present our design and implementation. In Chapter 11 we present our demo applications, and we evaluate the whole thesis in Chapter 12.

Chapter 5

Options

In this chapter we present some of the options we have for our design and implementation.

5.1 Main approaches to streaming

While reading literature relevant to this thesis, I came across two major approaches to dealing with the issue of streaming in distributed computing.

Existing frameworks, like CORBA [7] and ICE [11], make use of an *Interface Declaration Language* ***(IDL)*** as a means of creating a contract between clients and servers. Neither the CORBA IDL language nor Slice include any implicit support for streaming and multimedia, as they have not been created with this end in mind. Instead, the interfaces that can be declared in these IDLs are only implementing the RPC paradigm.

Researchers working on multimedia streaming issues in connection to existing (CORBA) ORBs[1], have therefore had to make a choice in regard to the use of IDL languages:

- to extend the existing IDL with concepts native to streaming, or
- to try to implement concepts native to streaming by means of traditional interfaces only.

In addition to these two main approaches, the MULTE project intended to try something new, which does not necessarily preclude the need to make a choice regarding to wheather to extend an IDL language or not. [15] states that *"Support for stream interactions need an extended IDL to specify stream interfaces with QoS specification for different flows. A stream object adapter supporting the generated stream stubs and skeletons will be developed. A new IDL compiler (back-end) targeted to Da CaPo will be used to produce stubs*

[1] As far as we know, our project is the first to use a non-CORBA ORB in multimedia research

(A-modules) integrating marshaling as part of Da CaPo's responsability and functionality."

What is new in the approach suggested by [15] is the desire to implement a *stream object adapter*, as an alternative to the common object adapter of ORBs.

In the remainder of this section, we discuss all these issues in more detail.

5.1.1 Extending an IDL

As an example of this approach we refer to the work presented in [1], where the designers of Sumo-ORB present a model of distributed computing which is extended with features supporting streaming. From their analysis of the requirements imposed by streaming multimedia data they derived the "necessary" extensions to CORBA's IDL and show how the language could be extended so that streams and flows can be treated like first class objects, just as interfaces are in the native CORBA IDL.

In short, they propose to extend the CORBA IDL to allow the declarations of *stream* and *signal* interfaces, in addition to the regular operational interfaces. This is according to the RM-ODP guidelines presented in Section 2.3. They propose the addition of a qualifier to interface declarations which, together with new keywords like flowing, flowOut, signalIn, signalOut, would tell the compiler what kind of code to generate for each interface. The following are examples of declarations from [1]:

```
interface <operational> cameraControl {
  start();
  stop();
  pan(in panDegrees integer);
  tilt(in tiltDegrees integer);
  zoom(in zoomFactor integer);
}

interface <stream> microphoneOut {
  flowOut audioOut(audio);
}

interface <signal> qosControl {
  signalOut audioSent(timestemp);
  signalIn audioDelievered(timestemp);
}
```

However, already in their early work they state their intention to bring their design up to *"compliance with existing CORBA implementations"*[2]. By this they mean to provide a solution based on pure CORBA IDL.

[2]See Section 11.1 of [1]

5.1.2 Using only CORBA IDL

As far as we understand, the work on the Sumo-ORB has been discontinued in favor for their newer multimedia middleware, TOAST [3]. In this later work, researchers from the same group have indeed aligned their work with the CORBA specification, by implementing an infrastructure of CORBA objects (operations), declared entirely in CORBA IDL. When these operations are implemented they will provide streaming services. They say that a *"notable feature"* of TOAST is that the middleware is *"designed and specified completely in terms of CORBA IDL"*[3].

This newer approach has evident advantages: the TOAST implementation should compile with any ORB, just as a C++ program should compile with any C++ compiler.

5.1.3 A streaming object adapter

Frameworks for distributed programming include the concept of an *Object Adapter* ***(OA)*** as the means to provide access to sets of related services provided by the framework.

Both ICE and CORBA have object adapters. They both have been designed to implement the RPC communication paradigm, and they both employ the object adapter for basically the same purposes, as presented in Sections 2.4.2 and 3.2.11. However, this communication pattern has proven to be of limited use for multimedia distributed programming because of the continuous nature of multimedia data and because multimedia data often comes in so large quantities that it can not be processed as a regular *operation*.

These reasons have been considered by the MULTE-ORB researchers to be so weighty as to warrant the introduction of a new object adapter. Remember from Section 2.4.2 that [7] advises to consider the introduction of new object adapters only when *"radically different services or interfaces"* are needed.

5.2 ORB based programming styles

In general, in a distributed application we want to separate the concept of *data* into *control data* and *payload data*. Control data is used to send the values of control parameters back and forth from the client to the server, in order to regulate the interaction between them. Payload data is the data which we are mainly interested in and whose transfer is regulated with control operations, which use control data. Multimedia data in a multimedia distributed application will be part or all of the application's payload data.

[3] See Section 3.2 of [3]

5.2.1 ORBs and sockets

The traditional way of building multimedia distributed applications has been to establish separated physical *paths* for the control and the payload data. The control path is usually established over TCP, because of its automatic retransmission of lost packets and its ordered delievery of packets. The path for the payload data can be established over UDP too, if occasional lost of packets, or occasional unordered delievery of packets is not critical to the application. The A/V Streams Specification imposes such a distinction between the control and the payload paths, where the control path is established by means of a CORBA ORB, and the payload path is established using regular socket mechanisms. Figure 5.1 shows the generalized scenario of separated control and payload paths. The fact that any ORB uses sockets internally, is not relevant for this discussion.

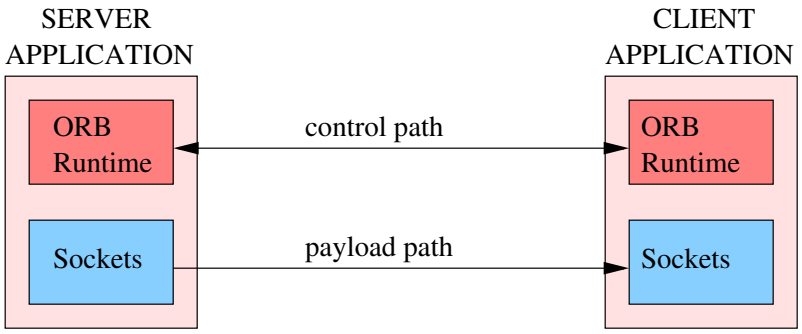

Figure 5.1: *Different mechanisms for control and payload data paths.*

5.2.2 ORBs only: *polling* and *callback*

The polling client and the callback programing styles discussed here are scenarios in which we still implement separated paths for the control and the payload data, but we direct both of them through the ORB. Figure 5.2 depicts this graphically.

The obvious disadvantage of directing all communication through the ORB is that the flow of payload data must be modeled by means of operations. This implies more messages exchanged between the ORB run-times on the client and the server end-points. More messages lead to more processing time on each end-point. If the operations employed are two-way operations, an additional delay is introduced, as the application has to wait for the reply, and it takes at least a trip-time for it to arrive.

The advantage of the approach is that all communication can be modeled by means of well defined interfaces and operations. This feature is especially useful for an automation task as the one we present in this thesis, because

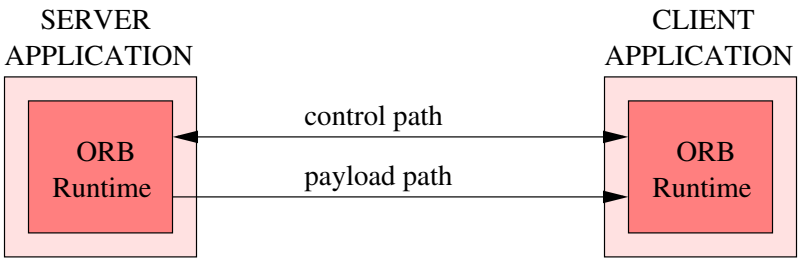

Figure 5.2: *Control and payload data paths through the ORB.*

interfaces and operations can easily be generated programmatically. Any distributed platform will guarantee the correct behavior of interfaces defined in its native IDL, and will also provide location transparency for end-points.

In this approach, both control messages and the transfer of payload data is realized by means of operations. In this thesis, interfaces, and therefore also operations, are always regular, as opposed to *stream* or *signal* operations which have been presented in Section 2.3. We recall also that some proeminent researchers have abandoned the use of stream and signal interfaces in their newer work, as discussed in Section 5.1.

Polling client

What is typical for a polling client application, is that the transfer of payload data is realized by means of *operations with return types*. The client calls the operations, to indicate that it wants payload data, and then collects the return result of each operation in a variable of an appropriate type. Thus, the client polls the server for each piece of payload data it needs. In this programming style, the client is the *active* part in the communication, while the server is *passive*, at least as far as the payload data it concerned.

The general blue print of *polling* operations is:

```
return_type operation_name(optional_parameters);
```

The client application will typically loop and call one or more operations in each iteration and do something with each piece of data which arrives. For an operation as the one above, the typical code would be:

```
return_type variable = 0;
loop {
  variable = operation_name(optional_parameters);
  use_data(variable);
}
```

Figure 5.3 shows the typical communication pattern which takes place for the transfer of payload data in a typical client polling application. This

pattern happens after the client and the server have been initialized and they have been connected to each other.

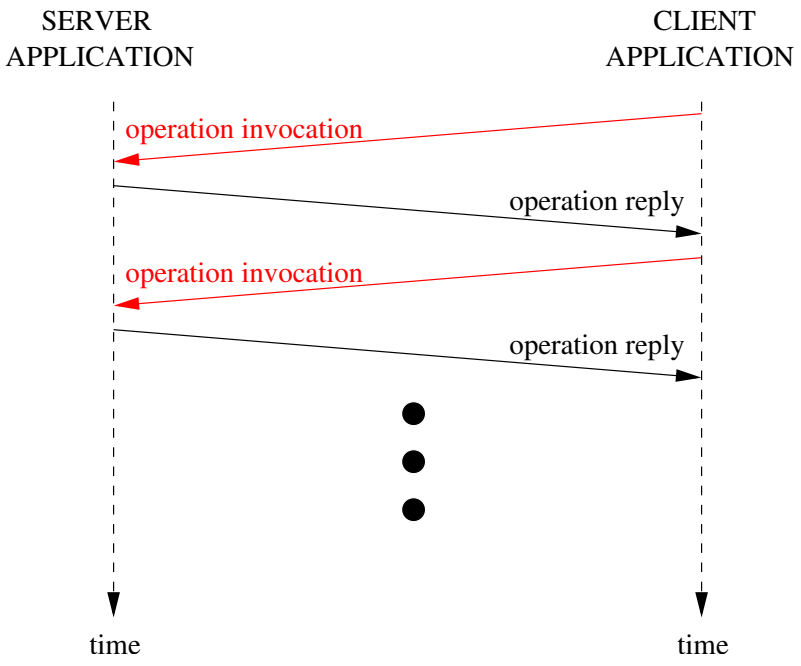

Figure 5.3: *Polling application communication pattern for payload data*

Both operation invocations and operation replies take time, an average trip time, and for the replies we have to add the processing time needed by the the server to produce the data which the client has requested. While the operation replies can be considered as an inevitable time expense, after all the data must be transfered to the client no matter what programming style is used, the operation invocations, shown in red in Figure 5.3, are clearly time expenses induced by this particular approach. The advantage of the polling client method is that it is simple to implement.

In the typical polling application, there is no need for a special binding object, as mentioned in the general object model of RM-ODP, which we have discussed in Section 2.3. The *contract* expressed by the declaration of operations assures that call to operations are automatically directed to the right servant objects, and if the operations return values, that the return values are sent back to the right operation calling objects.

Callback server

The callback server approach eliminates the time penalty associated with the polling client programming style. In this approach, "the client" continues to play the role of a client as far as the consumption of the data is concerned.

The client still is the application which needs the data which a server has. However, when it comes to the process of transporting the data to the client, the client and the server reverse their roles.

The transportation of the data is still accomplished by invoking operations, but this time it is the server who invokes *data carrying* operations on the client. Such a data carrying operation does not need a return type, and should not have one, unless a confirmation for the successful reception of each single packet of data is necessary. If the carrying operation had a return type it would invoke the same time penalty which we have seen in the case of the polling client programing style.

What makes this approach implementable is that two sets of operations are declared: one which the client invokes on the server, as in any regular distributed application, and another set with operations which the server invokes on the client, which we informally call *data carrying* operations.

The typical blue print of a data carrying operation, is therefore:

```
void callback(data_type variable_name);
```

In a server callback application, the client is the active entity only as far as initiating the transfer of data is concerned. After it has been prompted by a client to send data, the server becomes the active entity as it will continually callback the data carrying operation on the client, without waiting for any replies from the client. Each time the operation is "executed" in the client application, the client will simply collect the data sent as a parameter to the operation and process it at will. Figure 5.4 shows the communication pattern of a server callback based application, after all initialization is done.

There are no red arrows in this figure, because the client's notification to the server that it is ready to receive data, represented by the blue arrow in this figure, is an absolute necessity. We see that this approach does not incur a time penalty, because the data carrying operations can be conceived to correspond to the transportation of the data by means of any socket mechanism. Of course a little more processing is involved in this approach, compared to a pure sockets implementation, but the time needed for the data transfer itself will always be "much" larger then the processing of messages inside ORBs, because multimedia data is so voluminous.

Programming a server callback application requires the creation and initialization an object adapter in each of the client and the server applications. Each of the object adapters will dispatch operation calls coming from the other application than the one to which it belongs. This makes the initialization of a server callback based application more complicated than the initialization of a client polling application. In this case both applications must be informed about the proxies they can use to invoke operations on each other, whereas in the polling client application only the client needed be given a proxy to the server.

Figure 5.5 shows the *main* events which take place in order to achieve

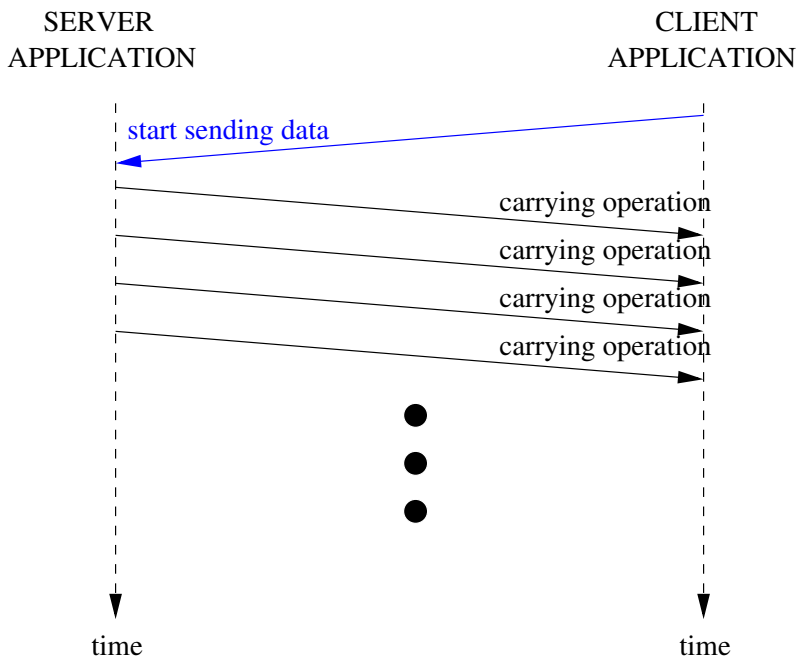

Figure 5.4: *Callback application communication pattern for payload data*

this.

First, the object adapter is initialized in both the client and the server. Then, other objects are initialized on both sides of the application. Especially important is a proxy to the one of the client's own object implementations. The client can easily produce this proxy by itself, by means of its own object adapter. The client application must then be given a proxy to a server based object implementation – this step is the typical ORB initialization "magic". After the client has this proxy, it can call an operation on the server, whose main purpose is to give the server a proxy to one of its own object implementations. The server stores this referance and acknowledges its reception to the client. From this time on, represented by the dotted horizontal line in Figure 5.5, the communication pattern represented in Figure 5.4 can begin to take place.

5.3 Compiler tools (PLY)

In the early stage of the work with this thesis, we considered to use the same compiler generator which has been used for FIDL in [16], called PCCTS [19, 20]. The main argument was to remain compatible with the work done in [16]. Also, at that time, PCCTS was considered to be one of the best compiler generators available. However, since we did not manage to

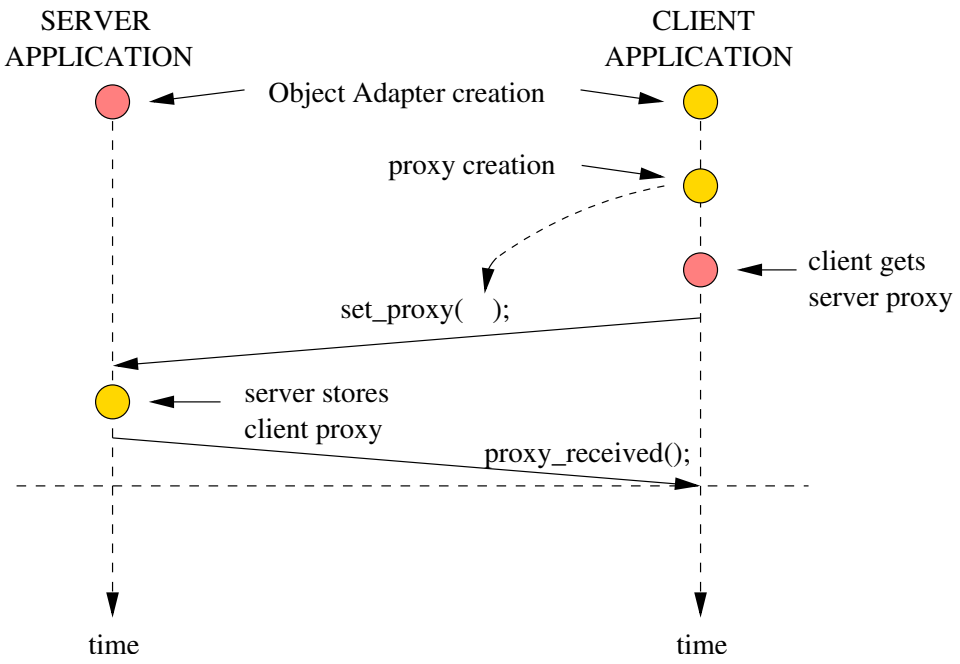

Figure 5.5: *Callback proxy hand-over.*

compile the code from [16], and since the PCCTS tool was not under development anymore, when this author came back from his external working assignments, we decided to look for alternatives. Our preference was to find a tool developed for Python, because Python is an excellent language for parsing.

We came across *Python Lex-Yacc* ***(PLY)*** [18], which is s a pure Python implementation of the well know LEX and YACC tools of Unix. This tool is developed at the University of Chicago, for educational purposes. We have used version 1.5.0 in this thesis. PLY is reported to be very easy to use and to provide very extensive error checking, which we can confirm. PLY comes with a 25 pages manual which is ideal to get you started if you have a basic understanding of grammars and of the process of writing parsers for a language. Otherwise, you will need to consult some introductory texts first.

Our experiences with PLY have been very good.

5.4 Decisions for this thesis

In the previous two sections we have presented alternative approaches to the issues IDL languages extended with streaming concepts and to ORB based multimedia programming styles. These are central issues in a work which attempts to extend an existing object-oriented middleware platform with

support for streaming.

For this thesis we choose to *not* extend Slice (the IDL of our chosen platform) with support for stream and signal interfaces. In this way, our work remains compatible with "any" version of ICE, because given the spread of use which ICE already has, ZeroC will be reluctant to make major changes to Slice's syntax and semantics for the forseeable future. This is an important feature of our solution, because, at least for the last year, ZeroC has released 4 versions of ICE. Maybe ICE will not be updated so often in the future, but even only one update per year would make our solution obsolete too soon, if it was making changes to Slice, unless of course, ZeroC would choose to integrate our extension into the system. Given the fact that others have moved from an extended IDL solution to a CORBA IDL compliant solution, we consider it to be wise to remain Slice compatible.

Regarding the original proposal to extend the very object adapter of an ORB, that would mean the object adapter of ICE in our case, we consider it a very good idea. It has the same disadvantage as extending Slice: our solution would have to be ported to every new release of ICE. In our oppinion it would be a nice and clean approach to offer streaming support at the level of the object adapter, but we also deem it to be too time consuming for us, given the fact that we have spent a considerable amount of time on Da CaPo. Probably the best way to pursue this line of thought is to set up another master's thesis's project, in cooperation with ZeroC.

We also choose to use FIDL as the means which offers that higher level of abstraction which is necessary in order to provide a descriptive representation of streams and flows of data. As we will see, we make small amendments to FIDL, in order to make it possible for our MSA to interface the environment into which it operates. We call our specification language FIDL++, even though the differences are so few that we wonder if they warrant the introduction of a new name. We also consider that it is a positive feature of our work to remain compatible with FIDL, because it is the language employed in the work presented in [1] and [25]. These implementations of an Intersector of flow specifications and of a Trader of multimedia gateways are features which our work can complement if they were to be integrated into a common system.

Finally, we choose to implement our MSA by means of the server callback approach. We opt to implement both control and payload functionality by means of ICE's run-time, i.e. an ORB only based solution, because we want to exploit to the maximum the transparencies offered by such a platform. We also believe that it is easier to model all communication by means of operations, instead of implementing our own application logic for the transfer of payload data. While Section 5.2.2 has given a general presentation of the server callback approach, Section 9.2 presents our ICE based implementation.

For the implementation of the FIDL++ compiler we have chosen to use PLY.

5.5 Summary

In this chapter we have presented the most relevant options we had to make in regard to our further design and implementation work. The options were identified as such from the literature studies done in connection with this thesis.

In the next chapter we present the design and implementation of a Da CaPo core, which is the first of the systems we work with in this thesis.

Chapter 6

Da CaPo

In this chapter we present a new design and implementation of a Da CaPo core. We begin in Section 6.1 by some general remarks, which show how the new implementation relates to the old one. In Section 6.2 we present the design and implementation of the Da CaPo modules, which are the basic building blocks of Da CaPo. In Section 6.3 we discuss the interfaces required from all Da CaPo modules. Section 6.4 gives a summary of the whole chapter. Section 11.1 presents a demo application which demonstrates how our design is used in practice.

6.1 General remarks

A Da CaPo ***stream*** is made up of at least one flow. In Da CaPo terminology, a ***flow*** is a unidirectional data current traveling from a ***source*** end-point to the ***sink*** end-point or a distributed application. These concepts are different from the stream and flow concepts of FIDL, as we have seen in the presentation of FIDL.

Figure 6.1 shows the components that typically make up a Da CaPo flow.

The modules are the main building blocks of a Da CaPo application. We have the same three categories of modules in this implementation, as we had in the original one. They are represented by the squares marked with capital As, Cs and Ts, respectively.

In this figure we have buffers between the modules. We believe that most applications need buffers, but in some simple situations, the modules could be linked directly to each other. Most modules will have both an ***input*** and an ***output*** buffer.

Figure 6.2 shows one module on each of the source and the sink end-points and the direction of the data flow on each end-point, represented by the arrows. The direction of the data flow determines which buffer in the figure is the input buffer and which one is the output buffer. We note that on

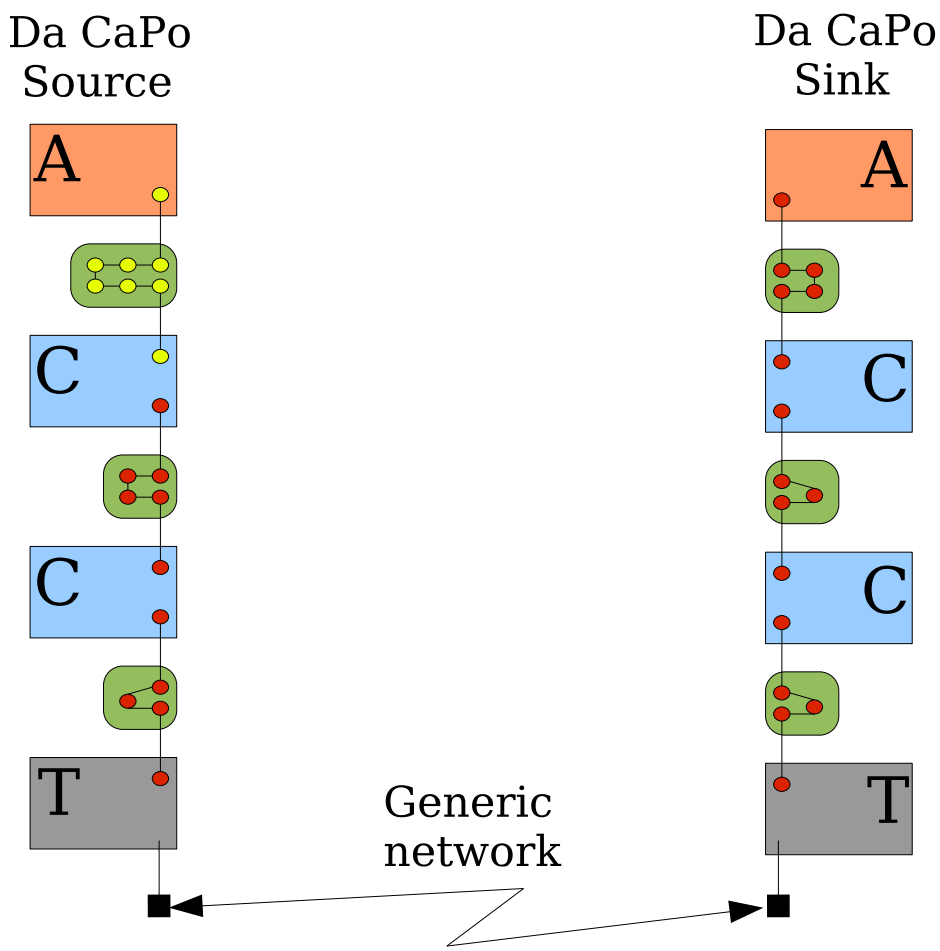

Figure 6.1: *Da CaPo flow.*

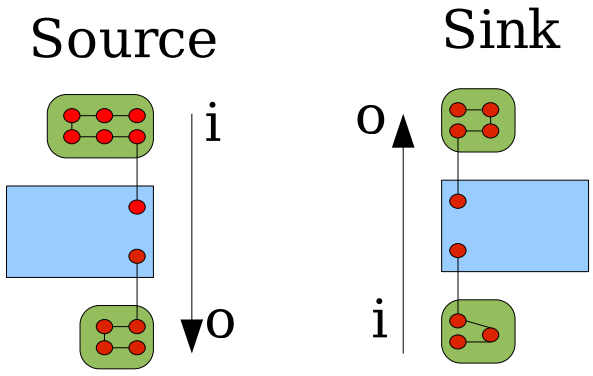

Figure 6.2: *Da CaPo module details: the direction of the data flow.*

the source node the input buffers are located above the modules and output buffers beneath them. On the sink node their location is reversed, because the data is traveling in the opposite (bottom-up) direction.

The module graph does not have to be symmetric on the two end-points. Most modules come naturally in pairs but they do not have to.

We can also see in Figure 6.1 that the number of packets in a buffer is independent of the number of packets in any other buffer in the whole Da CaPo application. This is so because in this figure we have depicted modules which are *independent* of each other. Each one executes in its own thread. As such, they produce and consume packets from their respective buffers as dictated by the thread scheduling mechanism which is employed by the operating system.

Even for modules which do pair up on the source and the sink end-points, there is no correlation between the number of packets in the buffers. Nobody knows exactly how long each packet will be delayed on its way from the source's T-module to the sink's T-module and the threading schedulers on the source and the sink end-points can not be expected to keep packets synchronized in the module graph.

In our opinion, to synchronize the processing of packets on the two end-points is not a necessary feature for most applications. Such a synchronization will considerably complicate the design of Da CaPo application and will introduce additional (execution) delays because of the extra checks which have to be executed.

When buffers show the same number of packets in our figures, they reflect a state of pure coincidence, which in practice will be the exception rather then the rule.

6.1.1 The A-module

The A-module represented by the orange rectangle marked with "A" in Figure 6.1, is the only module that generates Da CaPo packets on the source end-point and the only module that consumes the packets on the sink end-point. All intermediary modules will *process* the packets as they travel on their way to their destination.

The packets generated by the A-module in Figure 6.1 are represented by the yellow dots flowing downward from the module.

We note that the A-module on the source node only has an output buffer but no input buffer. This is natural because it is the module where the packets are generated from. On the sink side the A-module will only have an input buffer, because this module is the last one in the module graph; it is the module where the data is consumed. This corresponds to the fact that in the old implementation, the A-modules did not implement the whole of the *unified module interface*.

6.1.2 The C-module

Each C-module performs a specific networking function. In the most simple scenario a Da CaPo application will have no C-modules at all. The T-module will be linked directly to the A-module. On the source node the T-module's input buffer will be the A-module's output buffer, while on the sink node the T-module's output buffer will be the A-module's input buffer, as shown in Figure 6.3.

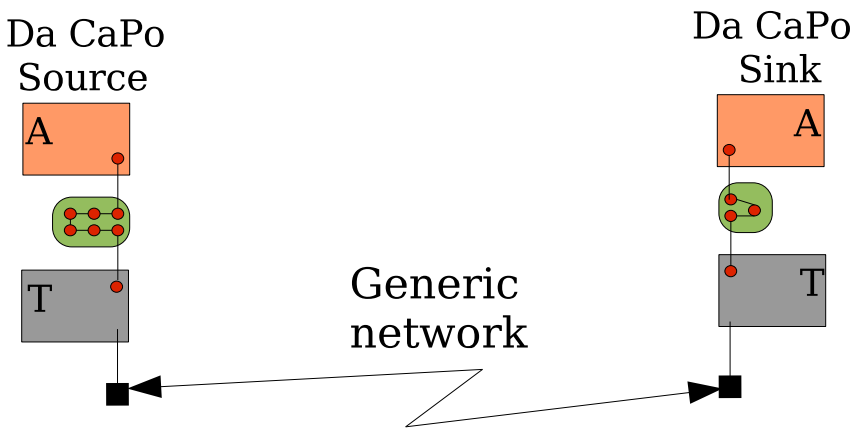

Figure 6.3: *Da CaPo flow: no C-modules.*

Just for the sake of argument, we note in Figure 6.1 that the packets generated from the A-module on the source side are represented by yellow circles, while the rest on the packets flowing in the system are represented by red circles. With this color distinction, which happens in connection with the first C-module on the source side, we try to emphasize the fact the this particular C-module is a module which alters the packets it receives in some *fundamental* way. Each module will perform some action on the packets it takes from its input buffer, but this doesn't have to result in a fundamental change to the packet. An example of what we mean with fundamental change, would be an encoding conversion performed by a module whose input packets contain image data in, say the YUVp420 encoding format, and whose output packets are images with another encoding, say JPEG.

6.1.3 The T-module

The T-module on the source side is responsible for delievering Da CaPo packets to the transport infrastructure. On the sink side, the T-module is responsible for retrieving the data from the transport infrastructure, reconstruct each Da CaPo packet and send them all further up in the module graph on the sink end-point.

It is therefore natural for the T-modules to have only a buffer, just as the A-modules do. We can say that the transport mechanism plays the role of the other buffer for the T-modules. We note that, on each end-point, the T-modules always have the opposite buffer than the A-module. On the source node the A-module has only an output buffer and the T-module has only an input buffer. The reverse is the case on the sink node: the A-module has only an input buffer and the T-module has only an output buffer. This is dictated by the direction in which the data flows through the module graph.

6.2 Da CaPo modules

The modules are Da CaPo's most basic computational unit. We have developed a module class hierarchy, which we present in Figure 6.4. The arrows point from the base classes to the derived classes. The Module class, depicted without a shadow in this figure, is the only abstract class, in the strict sens of the word. However, we believe that the ModuleI class might be the only one from which objects will be implemented in applications.

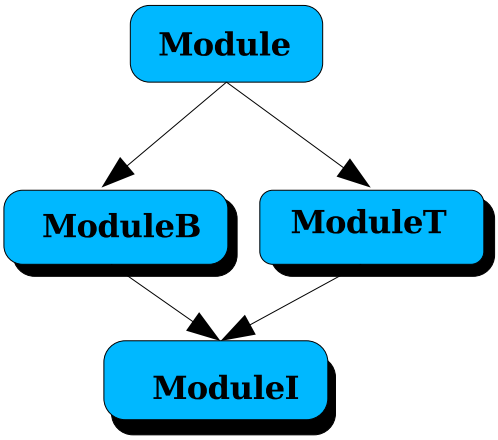

Figure 6.4: *Module class hierarchy.*

6.2.1 Module: the base module class

The Module class is a very simple abstract class, which is the base class for our module hierarchy. It is provided only for the purpose of being able to treat modules polymorphically in programs.. The Module class is declared as follows:

```
class Module {
 public:
  string name;
```

```
    int bytesProcessed;

    Module(string name);
};
```

6.2.2 ModuleB: the buffered (abstract) module class

This class adds buffer pointers to the Module class. Removing all irrelevant details, its declaration is as follows:

```
class ModuleB : virtual public Module {
 private:
 public:
  queue<DCPacket *> *inputBuffer;
  queue<DCPacket *> *outputBuffer;

  IceUtil::Mutex *inputMutex;
  IceUtil::Mutex *outputMutex;

  ModuleB(string name,
          queue<DCPacket *> *ib, queue<DCPacket
              *> *ob,
          IceUtil::Mutex *iM, IceUtil::Mutex *oM);
};
```

We implement buffers by means of *queues*, using the queue template class provided by the *Standard Template Library* ***(STL)*** of C++. A Da CaPo buffer is a regular FIFO queue.

For convenience's sake we have also defined DCBuffer in file buffer.h to be the same as a STL queue of pointers to Da CaPo packets:

```
typedef std::queue<DCPacket *> DCBuffer;
```

We note that the queues are not part of the buffered modules, but are only pointed to by them. Each module which derives from this class will read one packet at a time from its input queue, process the packet and write it to its output queue.

6.2.3 The Da CaPo packet

The elements which will be placed in the buffers are Da CaPo packets. A ***Da CaPo packet*** is *universal* in the sense that it can carry any kind of data. A suitable data structure for this purpose would be arrays of char or uchar. We have defined the DCPacket type in file buffer.h to be a substitute of char:

```
typedef char DCPacket;
```

In this way we can easily provide pointers to char by a construct like DCPacket *.

Byte(s)	Field Name	Size	Range
0	HeaderLength	1	0 to 255
1	FlowNumber	1	0 to 255
2	NumberOfFlows	1	0 to 255
3	FragmentNumber	1	0 to 255
4	ReferanceCount	1	0 to 255
5 to 7	unused		
8 to 11	SequenceNumber	4	0 to 2^{32} - 1
12 to 15	Payloadlength	4	0 to 2^{32} - 1

Table 6.1: The fields of a Da CaPo packet's header.

A Da CaPo packet is composed of two main parts: a ***header*** and a ***payload***. We have defined several ***header fields*** for this implementation, and have defined offset variables in buffer.h to point to each of the fields location within a packet. Currently the header is 16 bytes long. Table 6.1 presents the fields in the order in which they occur in the header, with some additional information about each of them.

We will discuss now the purpose of the header fields.

HeaderLength

An application might need to know the length of the header of Da CaPo packets so that it may know where the payload begins. This value is made available by the _DACAPO_PAYLOAD_OFFSET_ constant defined in buffer.h, but it is common to have this information within the header itself, so we have included it in the first byte.

FlowNumber

The FlowNumber field is useful in applications where there are several A-modules, at least on the producer side. An example of such an application would be a Da CaPo implementation of our demo application. Then we would have two kind of packets, one containing images, and the other containing sound, as our application would have two Da CaPo flows.

As each A-module produces it's packets, it will place them in its output buffers. It is likely that all the A-modules on the producer side will have one common output buffer. As packets travel through the Da CaPo graph, each successive C-module will read a packet from this common buffer and perform some action on it. Some C-modules will perform their actions on all packets, regardless of which Da CaPo flow they belong to. Other C-modules might process only packets belonging to certain flows and pass the other packets right through to their output buffer.

NumberOfFlows

A Da CaPo module might want to know how many flows are processed by the application it is a part of, so that it can iterate through all of them by means of for loops or switch statements.

FragmentNumber

In certain situations the need to fragment and defragment packets appears. For instance if a Da CaPo application uses the UDP transport protocol, and the A-module on the producer side generates packets larger then the maximum size of a UDP datagram (slightly more then 65KB), a fragmenter module must be inserted before the T-module interfacing UDP on the producer side and a defragmenter module must be inserted after the T-module interfacing UDP on the consumer side.

The combination of the SequenceNumber and the FragmentNumber fields will allow the defragmenter module to know which packets to merge together, in order to reconstruct the original Da CaPo packet.

ReferanceCount

A packet can be processed by *parallell* modules. A module might want to know if it is the only possessor of a packet at any given time. This becomes important for instance in modules where the packet is discharged after its payload has been consumed.

SequenceNumber

This field is used to differentiate between packets belonging to the same Da CaPo flow. It should be set by the A-module on the producer endpoint, and it should be only read by all other modules. Modules like those implementing ordered delievery of packets, for instance a pair of *Idle Repeat Request* modules used in conjection with an unreliable transport mechanism like UDP will rely on the value of this field to know how to process each incoming packet.

PayloadLength

In many circumstances, applications need to know the length of a piece of multimedia data in order to be able to consume it. This field should always contain the value of the length of its packet's own payload. In many situations, the packets will have the same payload length for long periods of time, maybe for the whole period of the execution, but it does not have to be like this. In our Da CaPo demo application the payload length changes every time the image size changes. Also, in application which handle several Da CaPo flows at the same time, the application should not be forced to deduce

the length of a packet by mapping the value of its FlowNumber field to a certain payload length. We can also think of situations where the payload length will vary almost with every packet, like an webcam based application which streams JPEG images, whose size in bytes vary with the content of the image, in spite of a fixed image size.

Reading and setting header fields

We have chosen to directly access the fields in packet headers. In order to set a field which is 1 byte long you will have to use code like the following:

```
DCPacket *dcp;
...
char flowNumber = 12;
memcpy(dcp+_DACAPO_FLOW_NUMBER_OFFSET_, &
    flowNumber, 1);
```

Here we set the flowNumber field to 12. Note that flowNumber is a char. In order to set a field which is size_of(int) bytes long, like SequenceNumber, you can execute code like the following:

```
DCPacket *dcp;
int seqNo;
...
seqNo++;
memcpy(dcp+_DACAPO_SEQUENCE_NUMBER_OFFSET_,
        &seqNo, _DACAPO_SIZE_OF_INT_);
```

You can read the value of a header field which is 1 byte long by executing code like this:

```
char nof = *(char*)(dcp+
    _DACAPO_NUMBER_OF_FRAGMENTS_OFFSET_);
```

And you can read the value of a header field which is size_of(int) bytes long by executing code like this:

```
int seqNo = *(int *)(dcp+
    _DACAPO_SEQUENCE_NUMBER_OFFSET_);
```

By redefining the header field constants in buffer.h, you can easily change the locations of fields in the header and add or remove fields without disrupting code written as in the examples given here.

6.2.4 ModuleT: the threaded (abstract) module class

This class adds to the Module class the ability for the modules to run in their own threads of execution. Modules derived from this class will be independent modules, as far as the threading model is concerned. The threaded module class is declared as follows:

```
class ModuleT : virtual public Module,
                public IceUtil::Thread {
 public:
  IceUtil::ThreadControl self;

  ModuleT(string name);
  virtual void run();
};
```

The ModuleT class inherits from the Thread class of ICE's own threading API. Since Da CaPo can be used in connection with the MSA it will be running in environments where ICE can run[1], so this is one way for us to provide operating system independent threading capabilities to Da CaPo.

6.2.5 ModuleI: the independent module class

The ModuleI class inherits from both the ModuleB and ModuleT classes, thus being both buffered and thread enabled. We have considered other kids of modules too, like bufferless, directly linked modules, but they seemed to us to be useful only in simple, special cases of the general streaming application. We consider that the buffered and threaded modules will provide the largest degree of flexibility to Da CaPo applications. The ModuleI class is declared as follows:

```
class ModuleI : public ModuleB, public ModuleT {
 public:
  ModuleI(string n,
          queue<DCPacket *> *ib, queue<DCPacket
              *> *ob,
          IceUtil::Mutex *iM, IceUtil::Mutex *oM);
};
```

As we can see, the constructor of this class will take as parameters the typical buffered module initialization parameters. Indeed, they are passed right down to the constructor of the ModuleB class.

We found that is is usefull to place a certain amount of self restraint on modules derived form this class, so we defined a constant value in file buffer.h for the maximum size allowed for a buffer:

```
static const int _DACAPO_YIELD_CPU_ = 23;
```

If a buffered and threaded module discovers that it's output buffer has reached this limit, it will yield the CPU because the module which consumes its packets is obviously slower then itself, or maybe malfunctioning. This self restraint is not implemented automatically by the ModuleI class, but must be implemented in the classes which derive from it.

[1] At the time of this writting, ICE runs on many Unix-like platforms and on recent versions of Windows, such as Windows 2000 and Windows XP.

6.3 Da CaPo module interface

The interface of Da CaPo modules in our implementation is very simple: it consists only of the constructor of the ModuleI class. This simple interface stands in sharp contrast to the original Da CaPo's unified module interface, mentioned in Section 3.1.4. It is enough for our core implementation, but might need to be extended when more functionality is added to Da CaPo. The Da CaPo demo application presented in Section 11.1 shows how we intend to use our Da CaPo core.

In general, this constructor has the following declaration, reproduced here from Section 6.2.5:

```
ModuleI(string n,
        queue<DCPacket *> *ib, queue<DCPacket
            *> *ob,
        IceUtil::Mutex *iM, IceUtil::Mutex *oM);
```

The last four parameters, ib, ob, iM and oM stand for *input buffer*, *output buffer*, *input mutex* and *output mutex*. What can possibly vary it that classes which inherit the ModuleI class can add several other parameters to the ones required by this constructor. The modules we have implemented for our demo application do just that.

The general algorithm for creating module graphs by means of this simple interface is:

1. Place the modules vertically, one under the other, in the order in which they are to be linked, for the producer side and for the consumer side, as depicted in Figure 6.1 and Figure 6.3.

2. For every group of two consequent modules, declare and initialize a buffer variable and a mutex variable, for the producer side and for the consumer side.

3. Considering that each of these buffers must be logically placed between two modules, invoke the constructors of your modules' classes and pass to them pointers to the buffer and mutex variables in such a way that the buffer will become the output buffer of the module above itself on the producer side and beneath itself on the consumer side, and the input buffer for the module beneath itself on the producer side and above itself on the consumer side. What we mean with output and input buffers has been explained in Section 6.1, and is depicted graphically in Figure 6.2. The mutex variables must always be given as parameters to the constructors in the same way as the buffers.

   Since the A-modules and the T-modules do not have both input and output buffers, pass 0 as a parameter instead of the missing buffer and mutexes.

By convention, all class names for the classes which implement modules begin with a capital M.

6.4 Summary

The modules are the basic building blocks of Da CaPo. We have begun this chapter by showing how our design of a Da CaPo core uses the same categories of modules as the old implementation of Da CaPo. Then we presented the C++ classes we have defined to implement these modules. The modules' interface consists in our implementation of only the constructor of the Modulel class. The algorithm presented in Section 6.3 shows how to write code which implements a Da CaPo module graph for both the producer and the consumer sides of an application. Section 11.1 presents in more detail how to build a Da CaPo based application.

The next chapter presents the design and implementation of our next subsystem, the FIDL++ language.

Chapter 7

FIDL++

A specification language like Slice or CORBA IDL provides a way to express interaction patterns which are not easily expressed in regular programming languages. The declarations made in such a specification language are then translated by special compilers into program statements in a regular programming language. When these program statements are executed they will accomplish the desired interaction behavior. Such a specification language provides therefore an higher abstraction level than a regular programming language can provide.

In this thesis, we attempt to provide an even higher abstraction level than that provided by Slice or CORBA IDL. Our goal is to make the abstractions of streams and flows available to the programmer, as regular C++ objects, in the context of object-oriented middleware.

Just as the higher programming abstractions provided by middleware for distributed programming are expressed in special purpose languages, we also will attempt to achieve an even higher level of abstraction by means of FIDL++, which in this context is another specification language.

We have implemented a FIDL++ to Slice and C++ compiler, and we will present in this chapter its design and implementation. In order to argue for the design decisions made for our compiler we refer often to the presentation of FIDL given in Section 3.3. When necessary, we also supplement that presentation with details which we felt did not naturally fit into such a general presentation of the language, as given there.

7.1 The formal background of FIDL

FIDL has been designed on a strong formal background. It is therefore a powerful language which is both concise and expressive at the same time. The strong analytic background of the language is not in the focus of this thesis at all, so we will not dwell on it, but this background is one of the reasons which has made us opt for this language. As our design of FIDL++

shows, we have tried to remain compatible with the original FIDL language.

As any other computer language grammar, the grammar of FIDL can not both allow much freedom and impose unique interpretations for all the language's features.

We discuss here what restrictions we have implemented in our compilers. It is likely that also the system implemented in [16] has imposed some restrictions to the many possibilities theoretically allowed by the grammar.

7.2 The concepts of streams and flows

As we have seen, FIDL allows the programmer to define streams. A stream is made of one or several flows. A flow in its turn is made of one or several media elements. From the examples given in [16] it is clear to us that the media element is the unit at whose granularity the data is generated and consumed in FIDL based systems.

We have chosen to graphically depict this aggregation of concepts as shown in the following figure. The white cylinder represents a stream of data.

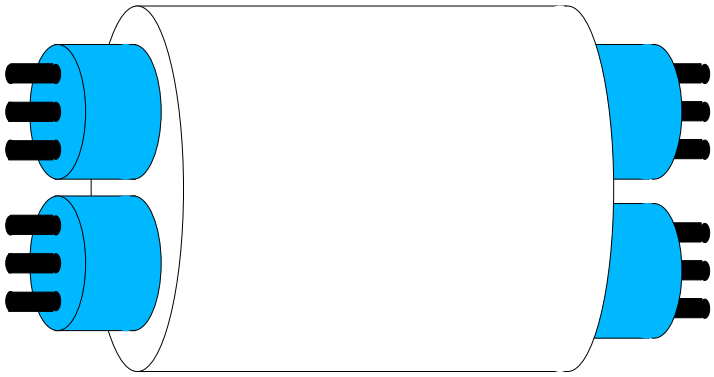

Figure 7.1: *The FIDL stream concept*

In this figure, the stream is composed of two flows. These are represented by the light blue cylinders. Each of the two flows is composed by three elements, represented by the black cylinders. The data flows through the black cylinders, and the other cylinders provide only higher levels of logical encapsulation.

This three level logical division is one of the things that we do not really understand about FIDL. For all practical means it seems that there is one level too much, and we consider that it would be more natural to conceive the concepts of streams and flows as depicted here:

In this figure, we have removed one level of abstraction. In terms of the grammar and the syntax of the specification language, the concept of a media element dissapears and the media element attributes will be declared

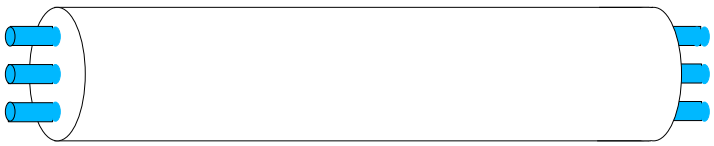

Figure 7.2: *An alternative stream concept*

within the flow construct. The data will then flow at the level of a flow, and a stream would simply be a collection of flows.

The reason why we do not see the necessity to have this one extra layer of abstraction is that on an end-point we will have at run time only one single FIDL stream object, which contains only one single FIDL flow object which is made up by one or more FIDL medial elements, as dictated by the constraint clause. Therefore the first two layers of abstraction (the stream and the flow objects) make each other redundant, and only one of them would suffice.

However, for the sake of remaining compatible with the previous work, and because this extra layer does not hinder us in any significant degree to illustrate how we have thought to provide access to first class stream objects in distributes applications, we have chosen to implement a compiler for the original FIDL language. Because the stream and the flow objects in a FIDL based application become redundant to each other upon implementation, it makes it difficult to differentiate between them in the writing of this thesis. In most cases they will refer to each other, and we could formulate the same thought using either of them. We consider this to be a weakness, because using a precise terminology for the description of a system is a very important requirement. Computer systems are usually complex enough in themselves, and a confusing terminology only makes things even harder to understand.

What we experience is that the data which is transported from one element to another is actually *flowing* from one element of an end-point to another element on a (potentially different) end-point, and thus forms a *flow*. Therefore, we also feel that it would be natural to also refer to it as a *flow*. But flow has already other meanings, among them being the second logical entity in FIDL's concept scheme of streaming. This is why we mean that it would have been better to eliminate the concepts of elements and only have streams of the simple kind presented in Figure 7.2.

7.3 The concept of media elements

The media elements in FIDL are what the flow specifications are made of (together with a constraint clause).

Each media element specification is a description of the characteristics, or

the qualities of a sing flow of data which we want our application to handle. A media element description is made of a list of attribute assignments.

The media classification and the attributes which have been proposed for each class, see Figure 3.5, are not necessarily final, as also the authors of [16] have pointed out.

This became evident also while working with our demo application. We experienced that we could fully describe the parameters of the sound part of the application, because the right attributes have already been introduced in FIDL. Therefore, it was sufficient for us to only declare one element of type audio, and include all the changes we want to be able to make to the quality of the sound in this description, by means of the values we have given to the various attributes. We repeat here the relevant part of the specification:

```
audio Sound {
  ...
  encoding = 2;
  channels = {1, 2};
  samplerate = {8000, 11025, 22050, 44100};
  samplesize = {8, 16};
};
```

The channels attribute is used to specify if the sound is to be mono or stereo. The default value is mono, because 1 is the first value on the right side of the assignment. The samplerate and the samplesize attributes are used to further decide the quality of the sound. Again, the default values are the first values declared on the right side of the assignments.

However, we could not easily fit in all the description of our video flow within one single media element. The problem we encountered was that one characteristic of an image, its *size* is actually described by two attributes: its width and its height. Since this was one of the characteristics whose value we wanted to be able to change during run-time, we have to specify several values for it at specification time. The only hindrance to that is that we do not want the width to be allowed to vary independently from the height. That would have been the result if we had opted to only declare one media element for images, like this:

```
image I {
  ...
  encoding = "YUVp420";
  samplerate = (5, 30);
  height = {120, 320, 640};
  width = {160, 240, 480};
};
```

With such a declaration it would have been perfectly legal to have at run-time an image of size, say 120x240. This is a size which our test hardware does not really support. We say *really support* because our hardware on the producer side, the webcam, is not so old and, as most hardware of its kind

nowadays, it supports "arbitrary" image sizes. This means that if the image size you request is not a "standard" image size, the hardware will choose a smaller image size then the one you have requested and fill in around this smaller image with a gray area up to the requested size. For the hardware on the consumer side, the screen, an arbitrary image size does not pose any special problems.

We had two choices in regard to this issue.

The first one was to introduce a new attribute name, say size, to the FIDL language. However, this solution would have posed a few problems of its own. First of all, what would be a legal value for the size attribute name? The most obvious answer is probably a string like "160x120". While this would be a feasible solution it would complicate the processing of the enhanced FIDL code, because such a value is rightly interpreted by the parser as a string value, while it's semantic meaning is a pair of integers. Even though we have not chosen this solution, we think it would be implementable. Another alternative would have been to give it as values sets of two integers, like size = {160, 120};, or even sets of multiples of two integers, in order to specify all the sizes at the same time. For instance size = {160, 120, 320, 240, 640, 480}; could be interpreted by the compiler to be three pairs of (width, height) tuples. But then we would have had to decide wheather size always must receive a set of multiples of double integers as values or only when it is declared for certain media element types. We also felt that we would not want to change the language more then absolutely necessary. As we will see later, we have made a very small addition to FIDL, but that is an addition which has a semantic meaning which has not been addressed by the original FIDL language, whereas adding more attribute names is not. Since the width and height attribute names are already present in the language, we decided to go for the second solution.

The second solution is to rather declare several media elements for our video needs, one for each image size. This has proven to be a fortunate choice, because it also gave us the opportunity to exercise other nice facets of FIDL, namely to make heavy use of the constraint clause of the language, as wee will see later in this report.

7.4 The intersection of flow specifications

In a distributed application there are at least two end-points involved. For each of the end-points we need to declare a stream specification which would describe this particular end-point's multimedia capabilities. Since end-points can run on heterogeneous hardware and software, the end-points will most often have different multimedia capabilities. The Intersector implemented in [16] is therefore a crucial component of the system because it is the module which reasons algorithmically about the end-points' multimedia capabilities

and present the system with an overview of the common capabilities. The system can then implement applications based on these common capabilities of the end-points.

In our solution, we depart a little bit from the common output of the FIDL Intersector. In program Listing 3.2, we have presented the output of the FIDL Intersector for our demo. As we do not interface the system implemented by [16] directly, we have chosen to use the format of the output of the FIDL Intersector as our starting point, but with a few modifications. Remember that the output of the FIDL Intersector is a *subset* of the FIDL syntax, and that the user defined identifiers are replaced with machine generated names. We have chosen to use the whole FIDL syntax and to make use of the user defined identifiers in our specifications. This will not pose any problem, as regular FIDL Intersector output will always be a subset of what our compiler can handle. If the two systems should be co-implemented our compiler will be able to process the output of the FIDL Intersector.

The real specification code which our compiler processes for our demo is therefore not the code shown in Listing 3.2, but the code we show here:

Listing 7.1: Intersector output in FIDL format for our demo's specification.

```
1  stream WebCamChat {
2    void flow F {
3      image QSIF {
4        producerElement = "EWebCam";
5        consumerElement = "EImageViewer";
6        encoding = "YUVp420";
7        samplerate = (5, 30);
8        height = 120;
9        width = 160;
10     };
11
12     image SIF {
13       producerElement = "EWebCam";
14       consumerElement = "EImageViewer";
15       encoding = "YUVp420";
16       samplerate = (5, 30);
17       height = 240;
18       width = 320;
19     };
20
21     image VGA {
22       producerElement = "EWebCam";
23       consumerElement = "EImageViewer";
24       encoding = "YUVp420";
25       samplerate = (5, 15);
26       height = 480;
27       width = 640;
28     };
```

```
29
30      audio Sound {
31        producerElement = "EMic";
32        consumerElement = "ESoundCard";
33        encoding = 2;
34        channels = {1, 2};
35        samplerate = {8000, 11025, 22050, 44100};
36        samplesize = {8, 16};
37      };
38
39      constraint (VGA | SIF | QSIF) & Sound;
40    };
41  }
```

7.5 Additions to FIDL

There is only one addition we have considered necessary to make to FIDL, in order to be able to implement all the features we have desired that our solution should have.

You might have noticed that in the code presented in the previous section we assign values to two attribute names which are not part of the original FIDL. These are the consumerElement and the producerElement attribute names. In general, the attributes provided by FIDL are enough for us to implement the internal structures which are necessary in order to offer the streaming API which we present in this thesis. Even though it is the networking which takes place behind the scenes in an application based on our solution that is the main issue of this thesis, there is another important feature that we wished to add. We also want to be able to connect the networking part of our implementation to the modules which implement the media elements. This is the purpose of these additional attributes. We will see later in this report how they are used.

7.6 The constraint clause

We consider the constraint clause of FIDL to be a very powerful and elegant way of expressing the exact composition of a stream in terms of media elements, which is the level where the data exchange takes place. The fact that a FIDL flow declaration contains several media elements, does not mean that all of these elements must be present in an incarnation of the flow. On the contrary, they only show which elements *can* take part in the implementation of the flow. This is illustrated also in our demo application. We have declared there three media elements for the video part of the application, one for each supported image size, and one for the audio part. This does not

mean that we want all four of these elements to be active all the time while we run our application.

The constraint clause allows us to tell the compiler what our intended use of the elements is. With the help of the *and* "&" and the *or* "|" operators, and with parenthesis we can specify *groups* of media elements. Each such group will be interpreted as a possible incarnation of the stream we are declaring. Since in this context a FIDL stream is the same as a FIDL flow, we will use the terms ***stream incarnation*** and ***flow implementation*** interchangeably.

In our compiler, we let the & operator have a higher precedence than the | operator. If parentheses are not used, all *and* operators will first bind their operands and then the results will be *or*ed together.

The constraint clause we have specified for our demo application

```
constraint (VGA | SIF | QSIF) & Sound;
```

will be interpreted to mean that a legal flow implementation for our stream will be made of any of the image elements together with the sound element.

More formally, the constraint clause will be translated to an ordered list of groups of elements. Each group represents a potential flow implementation. The list is formed by applying the two operators from left to right on the constraint clause. When we process the FIDL specification of our demo application, the parser will report the following:

```
Constraint interpretation:
--------------------------

1: ['VGA', 'Sound']
2: ['SIF', 'Sound']
3: ['QSIF', 'Sound']
```

Because it is often useful to have a uniquely ordered set of flow implementations, the constraint is interpreted from left to right, and the first flow implementation will be considered the most desired and the last one the least desired of them all.

Thus, we have specified in our constraint that in our demo application we mostly prefer to have VGA sized images and Sound. Our second priority is SIF sized images and Sound and our last priority is QSIF sized images and Sound. Based on this interpretation of the constraint clause, we will define the concept of QoS for our API. Before we do that, however, let us introduce the concept of *state* for a stream object as a whole and for lower level entities.

As we have seen in the presentation of the FIDL language, the constraint clause is optional, at least as far as FIDL's grammar is concerned. In the implementation of our compiler we require the presence of the constraint clause, even for the simplest stream specification. The reasons for this requirement are that it is easier to implement the internal logic of the compiler if we can assume that the constraint clause is always present and that we base our very definition of QoS levels on the interpretation of the constraint

clause, so we consider that it is important that the programmer has to think through what the constraint clause should be.

7.7 The concept of *state*

A stream object incarnates the concept of a stream of data. The data of a stream is a logical sum of the individual data of each media element which incarnates a particular flow implementation. Depending on how the data of each individual media element in a stream incarnation is flowing, we define several possible states. Figure 7.3 depicts graphically all the states a stream object can be in. Along the arrows we show which operations lead to state transitions in the stream objects.

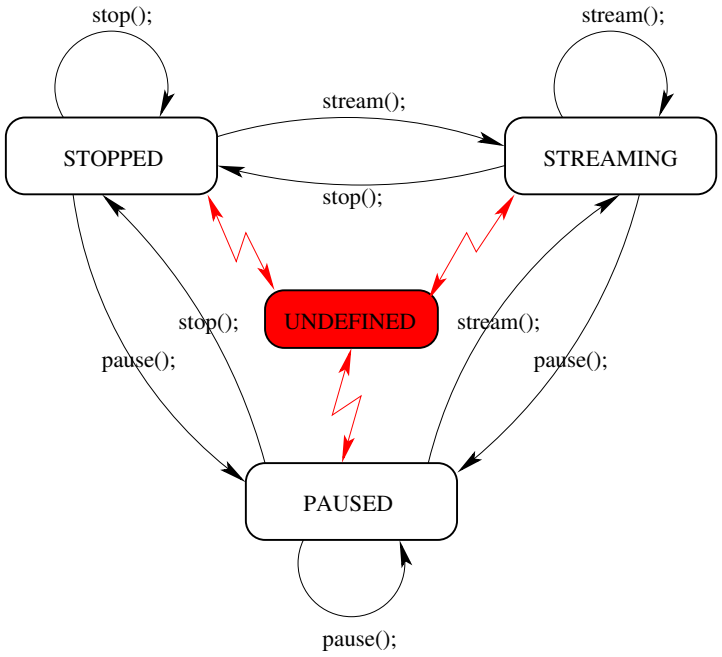

Figure 7.3: *A stream object's possible states.*

- **Stopped**: when a stream object is created, it is initialized to its Stopped state. This means that none of it's constituting media elements is sending or receiving data. A stream object will remain in its Stopped state until one of it's state altering operations is invoked on it. A stream object can be brought in and out of it's Stopped state any number of times.
- **Streaming**: a stream object is in the Streaming state if all of it's incarnation's media elements are flowing their particular data. A stream

object can be brought into its Streaming state by means of the API which we define in this thesis. A stream object can be brought in and out of its Streaming state any number of times.

- **Paused**: a stream object can be brought in the Paused state by means of API calls. When a stream object is Paused, none of it's media elements is flowing their particular data, just as when the stream object is Stopped. For some applications there will be no practical difference between a stream object's Paused and Stopped states. Typical for such applications is the fact that there is not a well defined *beginning* of the data they stream. Our demo is such an application. It will start streaming whenever the right API call is made, and the first data to be sent is generated at the time when the API call is made. Other applications, like a Video-On-Demand service have a definite meaning of the beginning of the data they stream. For these kind of applications, setting the stream object to the Paused state differs from setting it to the Stopped state in the fact that a Paused stream object remembers what data is next to be sent when it will be brought back to its Streaming state. After being Stopped a stream object will always begin Streaming from "the beginning" of its data.

- **Undefined**: a stream object is in an Undefined state is some of the elements of its active flow implementation are sending or receiving data 8depending on wheather they are on the sending or the receiving end-point), while others are not. A stream object is brought to its Undefined state by a failure of some kind, most likely by a networking failure. Some kind of mechanism should be developed to ensure that a stream object detects that it has been brought into the Undefined state so that it can attempt to rectify the situation. Ideally, the stream object should try to bring itself again into a legal Streaming state. If it does not succeed to do so, it should bring itself into the Stopped state. The reason why the Undefined state is unacceptable is that it gives no accurate description of what is working and what is broken among the stream object's composing elements. It is not possible to willingly bring a stream object into an Undefined state by API calls. For the sake of demonstration our compiler declares and implements a few "unhealthy" API calls which allow us to demonstrate how a stream object behaves in the Undefined state, but these API calls should not be generated in a production situation.

A stream object will ignore any attempts made to bring it by means of API calls into the same state into which it already is. This is an important optimization feature, because no unnecessary calls are dispatched across the network between the end-points.

7.8 The QoS concept

Since virtually all attributes which can be used to declare media elements are used to specify characteristics of the data to be streamed by each particular element[1], it is natural to consider the declaration of the media elements as declarations of levels of QoS. However, even though the data is streamed at the level of the media element, the constraint clause naturally binds together one or several media elements into logical flow implementations. Therefore, it is more natural to consider first of all the characteristics of a flow implementation as a QoS level.

Given this definition of a QoS level, we can say that a stream specification will have as many QoS levels as there are flow implementations.

Also, given the ordering of the flow implementations provided by the interpretation of the constraint clause, we can allow a simple mapping between the individual flow implementation and a QoS level to tell us what is the most preferred QoS level. We simply define that if the most preferred flow implementation is active, the stream object is operating at the most desired QoS level. If the least desired flow implementation is active, the stream object is operating at the least acceptable QoS level, and we use the same reasoning for all the flow implementations and QoS levels in between these two extremes.

In our API, we refer to a ***QoS level*** as ***QOSL*** followed by an integer value which specifies its ordering index. For our demo application, we have three flow implementations and therefore three QoS levels. If a stream object for our demo application is operating with VGA images and any kind of Sound, we say that the stream object is running at QOSL1. If the images provided are of SIF size and attended by any kind of Sound, we"ll say that the application runs at QOSL2 and, finally, if the images are of QSIF size and any kind of Sound is present we say that the application is running at QOSL3.

It is very important to notice that the QoS level is not the finest granularity at which we can address QoS. As an example, let us say that our demo application is running at QOSL1. The element specifications in use in this situation are declared as follows:

```
image VGA {
  producerElement = "EWebCam";
  consumerElement = "EImageViewer";
  encoding = "YUVp420";
  samplerate = (5, 15);
  height = 480;
  width = 640;
};
```

[1] The only exceptions are consumerElement and producerElement, the attributes we have added to FIDL.

```
audio Sound {
  producerElement = "EMic";
  consumerElement = "ESoundCard";
  encoding = 2;
  channels = {1, 2};
  samplerate = {8000, 11025, 22050, 44100};
  samplesize = {8, 16};
};
```

We can see from these declarations that both of them have at least one variable parameter, and therefore each of them can provide several kinds of data with different properties, and still fullfill the requirement of belonging to QOSL1.

For instance, the video data can be delievered at framerates between 5 and 15, but as long as the size of the images is VGA, the application will be running at QOSL1. This is because we have made the *size* of the image elements the criteria for moving from one QoS level to another and not the framerate. Therefore, we can say that there are sublevels of QoS within each QoS level.

Since the VGA media element has only one variable parameter, we could easily give unique names to each QoS sublevel. The samplerate parameter can have 11 different values, so we would have 11 QoS sublevels for QOSL1. Of course if the VGA element had several variable parameters we would have as many QoS sublevels as the product of the numbers of unique values each parameter can take. This is exemplified by the Sound media element which has three variable parameters with a total of $2 * 4 * 2 = 16$ combinations.

To determine the total number of QoS sublevels we have to also consider the fact that there can be several media elements in a flow implementation, and the total number of QoS sublevel will be the product of the numbers of QoS sublevel introduced by each media element.

Providing a good naming convention for so many sublevels of QoS levels, is not easy, because the names we have come up with become very long when there are many media elements in each flow implementation. Still, we consider that it is imperative to provide access for the programmer to every QoS sublevel. We have, therefore, chosen to address the issue of QoS sublevels indirectly, by providing access to each particular variable parameter. We return to this issue in Section 10.3.

7.9 Summary

In this chapter we have presented our interpretation of the FIDL++ language. Much of this discussion has been based on a comparison to the original FIDL language. We have discussed how the interpretation of the constraint clause of a FIDL++ specification is especially important. We

have also defined the states a stream object can be in, and how we relate the concept of QoS to FIDL++. In the next chapter we describe the process and the result of parsing FIDL++ specifications with our compiler.

Chapter 8

Code generation

In this chapter we describe the code which our compiler generates from FIDL++ descriptions. We look first at which files are generated, and what their purpose is, in Section 8.1. We continue then with a presentation of the individual classes which we use to implement the functionality of the MSA, in Section 8.2. In Section 8.3 we present the method we use to generate a workable interface between our MSA and the rest of the distributed application. Section 8.4 provides a summary of this chapter.

8.1 File names

The first step in the MSA programming cycle, is to provide a FIDL specification for the stream object we want to implement. We provide it in a file which, by convention, ends with the ".fdl" file name extension. In likeness with ICE and CORBA, our compiler will use the root of the file name for the main output file.

Thus, if we provide the code for our demo application in a file named webcam.fdl our compiler will generate a file named webcam.ice. This file will contain a Slice representation of the stream object we have declared in FIDL. This file must be compiled by ICE's compiler, slice2cpp. Following the file naming conventions of ICE, slice2cpp will generate two other files from webcam.ice. These will receive the webcam.h and webcam.cpp names.

Having a Slice representation of our stream object is a an important first step on our way toward the goal of having a functional C++ object available in our end-point applications. As we will see, this Slice representation declares types and operations which are necessary in order to implement a stream object. Still, these definitions are far from all what it takes to implement the desired stream object. Anyone who is familiar with the programming cycle used in middleware like ICE and CORBA knows that the declarations made in the middleware's declaration language must be implemented in the target programming language since the middleware will only

facilitate the networking that takes place between the end-points. The programmer must implement classes for the *proxies* and the *skeletons*, in ICE terminology, or the *stubs* and the *skeletons* in CORBA terminology. In fact, this often can be a more time consuming job then coding the declarations in the middleware's declaration language.

Therefore, our compiler will provide also the code which implements the proxies and the skeletons for the definitions for our stream object. This code is provided in two other files, named interfaceImplementations_callback.h and interfaceImplementations_callback.cpp. We could not continue to use only the root of the FIDL file's name, because ICE's file naming convention requires the webcam.h and webcam.cpp names for its own use.

So far we have not considered the scenario where our compiler is supposed to compile several FIDL files residing in the same directory. When such a need appears, it will probably be desirable to have the interface implementing code for each of the stream objects of each FIDL file in separate file. Our file naming policy should be changed then, so that files containing interface implementation code will not be overwritten. One solution is of course to use the root of the main file's name, but not on its own. For instance, if two FIDL files stars.fdl and planets.fdl are to be compiled in the same directory, the code which implements the interfaces for each of the files could be placed in files named iistars.h, iistars.cpp, iiplanets.h and iiplanets.cpp. The *ii* prefix is a shorthand version of *interface implementations*.

Even when all of these files are available, we still need to write code for the applications which will run on the end-points. Traditionally, these applications are called *server* and *client* in ICE terminology, in spite of the fact that it is not uncommon for an ICE application to behave like a client toward an applications and like a server toward another one. This terminology suits well our demo application, which is a one-way video-phone demo. The end-point where the webcam and the microphone reside can rightly be called *the server*, and the end-point where the data is consumed can rightly be called *the client*.

We generate also a basic server and a basic client application, in two files named serverCB and clientCB. These applications will initialize the ICE runtime in both the server and the client. This is something which must be done in all ICE based applications, from the simplest "Hello World!" demo to the most complex ones. We also create and initialize all the FIDL originated objects which are necessary for our streaming application to function. The only thing missing is the code for the application logic, but this is something which can not be computer generated.

Our server and client applications need appropriate configuration files for initialization, because we have not hardcoded all details in the applications themselves. Especially, the client applications's configuration file depends on the specific content of the FIDL specification from which it is generated. Our compiler will therefore generate one configuration file for each of the

server and the client applications. These files are called config.server.callback and config.client.callback.

Finally, our compiler also generates a Makefile which will compile all this code.

To summarize, given a FIDL specification file called webcam.fdl, our compiler will generate the following files from it:

1. webcam.ice
   Slice definitions to be compiled by slice2cpp.

2. interfaceImplementations_callback.h
   Header file for the C++ implementations of the Slice definitions from webcam.ice.

3. interfaceImplementations_callback.cpp
   Code file for the C++ implementations of the Slice definitions from webcam.ice.

4. serverCB.cpp
   Basic ICE server application. Fully functional, but the application logic can be extended.

5. clientCB.cpp
   Basic ICE client application. The application logic must be added to this file. Optionally, the client application can be hooked to a GUI.

6. config.server.callback
   Configuration file for the server application.

7. config.client.callback
   Configuration file for the client application.

8. Makefile
   Makefile to compile all of the above.

We want to mention that when all of these files are put together, the applications will be complete only in the sense that they will compile. As they are generated by our compiler, the server and the client executables will do nothing when they are run, even though they are fully initialized to start streaming according to the pattern described in the FIDL file from which they were generated. You must supply the application logic code, at least for the client application, and you may also want to hook at least the client application to a GUI interface. Of course you don't have to, but after all, the primary purpose of the MSA is to handle multimedia data and in most cases a GUI will be desirable. Our demo application, presented in Section 11.3, is an example of all of this, and we explain how to build a MSA based application from the ground up in Appendix A.

8.2 Class names

In the process of translating FIDL to Slice, our compiler will generate class names based on the declarations made in the FIDL file. We have tried to find a naming scheme for these classes which is both simple and intuitive. We will present our naming scheme in a top-down fashion.

The *"stream"* class

On the top of our class hierarchy is the class which implements the stream object. Since FIDL requires that a stream declaration should be given a name, we have chosen to use this name as the name for this class.

For our demo application we have made the follwing declaration:

```
stream WebCamChat {
   ...
}
```

Therefore, the class which implements the stream object in applications using this FIDL declaration is WebCamChat. Our compiler generates Slice definitions for this class in webcam.ice. The exact content of this definition depends on the rest of the FIDL declaration. After a Slice file is compiled by slice2cpp, someone must implement the definitions made in the Slice file. By convention, classes which implement Slice definitions are given the same name as the Slice definition followed by an "I", to signify that this is a class which *implements* a Slice definition. The class implementing the slice declaration of WebCamChat would therefore be named WebCamChatI. For the sake of better visibility we have chosen to use the "_I_"prefix instead of "I", in the code generated by our compiler. The class which will implement the WebCamChat declaration will therefore be called WebCamChat_I_. This class will be declared and defined in the interfaceImplementations_callback.h and interfaceImplementations_callback.cpp files, and will implement the Slice declaration of WebCamChat in webcam.ice.

An object of this class will be created and initialized in the clientCB.cpp file by the code generated by our compiler. We have chosen to name this object stream, because it is the object which incarnates the FIDL stream declaration. This means that for our demo application the compiler will generate the following line of code at an appropriate place in clientCB.cpp:

```
WebCamChat_I_ *stream = new WebCamChat_I_(...);
```

The list of parameters given to the constructor of this class depends on the contents of the FIDL declaration, and are omitted here. However, stream is a regular C++ pointer and the programmer can use it in the code of the application logic as any other pointers.

Producer side	Consumer side
PFIVGASound	CFIVGASound
PFISIFSound	CFISIFSound
PFIQSIFSound	CFIQSIFSound

Table 8.1: Slice classes for flow implementations.

The flow implementation classes

Even though a FIDL stream specification does not contain explicit declarations of flow implementations, they are deduced from the constraint clause of the specification and play a very important role in our solution. We declare therefore classes for the flow implementations also.

The interpretation of the constraint clause determines how many flow implementation classes there will be. In the case of our demo, the compiler will generate three of them. Since a flow implementation is made up of one or several medial elements it is natural to combine the names of the constituting media elements in order to create a unique flow implementation class name. Our demo applications had this constraint interpretation:

```
Constraint interpretation:
--------------------------

1: ['VGA', 'Sound']
2: ['SIF', 'Sound']
3: ['QSIF', 'Sound']
```

The preliminary *composite names* of the flow implementation classes are therefore VGASound, SIFSound and QSIFSound. Because we need flow implementation objects both on the producer and the consumer end-points, and because these objects will (indirectly) interface different hardware devices, we need a set of flow implementation classes for the producer side of the application and one set for the consumer side. We will therefore append the CFI and the PFI prefixes to the preliminary composite names mentioned above, and generate Slice declarations for two sets of flow implementation classes. The CFI prefix is a shorthand description meaning *consumer flow implementation* and PFI is a shorthand notations for *producer flow implementation*. The finale names for the flow implementation classes for our demo application are presented in Table 8.1.

This naming convention becomes quite verbose for flow implementations which are made up of more then two media elements, but since these objects are handled internally by our compiler, and not directly by the programmer, we do not mind so much.

First, we declare types for these flow implementation classes in webcam.ice. The exact contents of each of the flow implementation classes depends on the rest of the FIDL specification, but in general, each of them is an encapsulation of the Slice representations of the constituting media elements

Producer side	Consumer side
PFIVGASound_I_	CFIVGASound_I_
PFISIFSound_I_	CFISIFSound_I_
PFIQSIFSound_I_	CFIQSIFSound_I_

Table 8.2: C++ classes for flow implementations.

of the flow implementation.

Just as it was the case for the class implementing the stream declaration, we need to also implement these declarations of the flow implementation classes. Using the same naming convention, we declare and define the classes presented in Table 8.2, in the interfaceImplementations_callback.h and interfaceImplementations_callback.cppfiles.

The compiler will generate the code which will declare and initialize objects of these classes, in the serverCB.cpp and the clientCB.cpp files, as needed by the client and the server applications. The programmer is not hindered in any way to use and manipulate these objects directly, or to create additional objects of these classes, but this should not be necessary. If such a need appears in a particular setting, it probably signals that we need to extend the API of the MSA.

The QoS level binding classes

The QoS level binding classes are also an indirect product of the interpretation we have given to the constraint clause of a stream specification. In the subsection above we have presented the classes which define and implement the flow implementations, on both the producer and the client side of the application. These classes are logically connected, one class on the producer side having its peer on the consumer side of the application. In many practical ways, when data is streamed from the producer to the client in an application, objects belonging to such pairs of classes will work together, especially when control traffic is concerned.

Therefore, we see that, for all practical purposes, an object implementing a flow implementation on the producer side must be *binded* at runtime to its peer object on the consumer side. In order to facilitate this binding, we generate classes which will accomplish it. In keeping with the rest of our naming conventions, we generate names for these classes by using the QOSBindingLevel root, followed by an integer which specifies which flow implementation objects this class will bind. Remember from Section 7.8 that a QoS level is a synonym for a flow implementation.

For our demo application, the compiler will generate declarations in webcam.ice for the classes presented in the left column of Table 8.3. Each of these classes will implement a logical binding of a PFI and a CFI class, as shown in the right column of Table 8.3.

QoS level binding class	Paired classes
QOSBindingLevel1	PFIVGASound, CFIVGASound
QOSBindingLevel2	PFISIFSound, CFISIFSound
QOSBindingLevel3	PFIQSIFSound, CFIQSIFSound

Table 8.3: Slice QoS level binding classes.

_I_ QoS level binding class	Paired classes
QOSBindingLevel1_I_	PFIVGASound_I_, CFIVGASound_I_
QOSBindingLevel2_I_	PFISIFSound_I_, CFISIFSound_I_
QOSBindingLevel3_I_	PFIQSIFSound_I_, CFIQSIFSound_I_

Table 8.4: _I_ QoS level binding classes.

Just as for all the other Slice defined classes, we need to declare and define _I_ classes for the QoS level binding classes, which will implement them. The compiler will declare and define in interfaceImplementations_callback.h and interfaceImplementations_callback.cpp the classes presented in the left column of Table 8.4. These implementing classes will also provide a logical binding of CFI and PFI implementing classes, just as it was the case for the Slice classes. In the right column of Table 8.4, we show which classes are logically binded by the implementation QoS level binding classes.

The compiler will also generate code which declares and initializes objects of these classes in the serverCB.cpp and the clientCB.cpp files, as needed by the client and the server applications. Again, the programmer can manipulate these objects directly, and can instantiate new objects of these classes, but this should not be necessary. If the need to manipulate these objects directly appears, we should reconsider the design of our API.

The element classes

We have finally reached the bottom of our class hierarchy. A FIDL stream specification will contain at least one media element declaration. Some or all of these media element declarations will be used in the constraint clause. It is not a requirement that every media element declaration must be part of at least one flow implementation, but it is pointless to declare media element specifications which are not used in the constraint clause.

Each media element declaration must declare which modules are implementing its consumer side and its producer side functionality. For this purpose we use the the producerElement and the consumerElement attribute names which we have added to FIDL, as mentioned in Section 7.5.

Thus, for each media element declaration, we will have to interface two modules. Strictly speaking, we do not require that the two modules must be different, but we can not conceive yet any situation where it would be

Producer side	Consumer side
PECVGA	CECVGA
PECSIF	CECSIF
PECQSIF	CECQSIF
PECSound	CECSound

Table 8.5: Slice classes for media elements.

Producer side	Consumer side
PECVGA_I_	CECVGA_I_
PECSIF_I_	CECSIF_I_
PECQSIF_I_	CECQSIF_I_
PECSound_I_	CECSound_I_

Table 8.6: C++ classes for media elements.

useful to have media elements which have the same producer and consumer module.

For each media element declaration we need at runtime an object which will interface the modules which implement the producer and the consumer functionality of the media element.

To fullfill this need, the compiler will generate Slice declarations for media elements in webcam.ice. Since the media elements are uniquely identified by the names given to them in the FIDL specification, it is most natural to use these names as the names of the declared classes. However, since we need to differentiate the interfacing of the producer side module from the interfacing of the consumer side module, we need to declare two classes for each media element. We have chosen to create unique names for these classes by combining the unique media element names with the PEC and CEC prefixes. These prefixes are shorthand notations of *producer element class* and *consumer element class*.

In the specification of our demo application, we have declared a total of four media elements. The compiler will generate Slice declarations for all the classes presented in Table 8.5. The exact content of these declarations depends on which attributes have been set for each media element.

Also for these classes we need to declare and define C++ classes which will implement them. The compiler will therefore declare and define the classes presented in Table 8.6 in interfaceImplementations_callback.h and interfaceImplementations_callback.cpp:

The compiler will also generate code in the serverCB.cpp and clientCB.cpp files which declares and initializes objects of these types, as needed by the client and the server applications. Also these objects can be manipulated directly, and new objects of these types can be declared by the programmer, but this should not be necessary.

8.3 Media element modules

The subject we treat in this section is indirectly connected to code generation, so we treat it in this chapter.

A *media element module* ***(MEM)*** is a software module which is used to implement the functionality of a FIDL++ media element declaration. FIDL++ is a declarative language, and does not implement networking functionality or processing of multimedia data. A MEM is conceptually different than a Da CaPo module. In a scenario where Da CaPo and the MSA are co-implemented, the two concepts would overlap in many respects, but not otherwise.

MEMs are not part of the MSA presented in this thesis, but are provided for the sake of demonstrating the functionality of our MSA. For instance, our MSA demo application needs 4 MEMs: one which interfaces the webcam device, one which interfaces the microphone, one which is responsible for displaying the images and one which is responsible for playing the sound which comes from the microphone. Our compiler generates a *MEMs interface* for the MSA. This imposes certain requirements to the design of MEMs.

The names of MEMs can be whatever the programmer desires. We have chosen the convention of starting their names with a capital "E", because they provide the multimedia processing capability of the *element* declarations of any FIDL++ specification. For each media element declaration, we require that the consumerElement and the producerElement attributes must be declared. Thus, when a media element declaration is encountered, our compiler will pick up the values of these attributes and will use them as names of C++ objects. From a declaration like

```
image VGA {
  producerElement = "EWebCam";
  consumerElement = "EImageViewer";
  encoding = "YUVp420";
  samplerate = (5, 15);
  height = 480;
  width = 640;
};
```

the compiler will generate code which polls an object called EWebCam for VGA sized images, and which delivers the images thus provided to an object called EImageViewer. The MSA is not concerned with how the EWebCam object produces the images, and with what the EImageViewer objects does with them.

However, our compiler must know where the MEMs implementations are located, so that it can include their header files in the code it generates for the MSA. Also, while the compiler knows the names of the objects, it must also know the name of the files in which they are implemented. The name of the files could be the same as the names of the modules, only succeeded by

the .h and .cpp suffixes. However, we did want to separate these file names from other file names which by chance would also begin with a capital "E", so we require that a module must be implemented in files which prepend the module name with the e_ prefix. Thus a module called EModule must be implemented in the e_EModule.h and the e_EModule.cpp files.

As far as the location of the files is concerned, our compiler expects to find all module files in the base directory of the source code.

If we consider again the FIDL specification for our demo application, see Programme Listing 7.1, we can see that our media element declarations use two producer MEMs and two consumer MEMs to implement their functionality. This makes a total of four MEMs, and we see that a MEM can be used by several medial element declarations. The relationship between the MSA and the MEMs is depicted graphically in Figure 8.1, which shows the scenario for our MSA demo application.

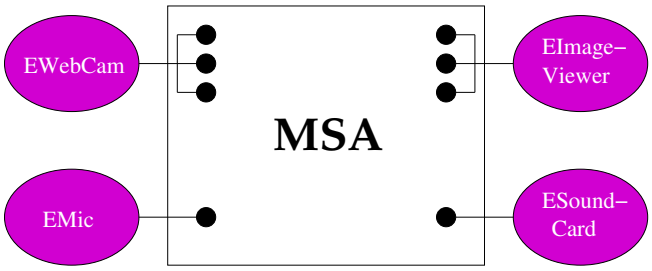

Figure 8.1: *Media module elements and the MSA.*

In addition to requirements to file locations and file naming, the MEMs' interface must *mirror* the media element declarations for which they are implemented. By mirroring we mean that if a media element declares an attribute to be variable (its value is not atomic), then both MEMs which implement its data processing functionality must declare a method corresponding to the interface defined in Section 10.3, for setting this attribute's value. This must be so, because when we make an API call like

```
stream.setEWebCamsamplerate(15);
```

the call will eventually be dispatched down to the appropriate MEMs, since they are the only objects which actually really can execute it.

The fact that the MEMs are required to support this particular interface should not be a problem, because the MEMs implementors usually are the same people which also write the FIDL++ specification.

8.4 Summary

The main contribution of this chapter is an explanation of the code we have considered necessary to generate in order to implement a MSA from a FIDL++ specification. We have therefore considered, both how we have organized this code into files, and the naming conventions used for the classes we generate. Finally, we have presented the requirements posed to MEMs, in order for our system to be able to interface them.

In the next chapter we present how we use the code explained in this chapter, by looking at the run-time of a MSA based application.

Chapter 9

The MSA run-time

In this chapter we describe the run-time environment of a MSA based application. Since the main requirement posed to our MSA is that it should operate within object-oriented middleware, and given the fact that we have chosen to use ICE as our implementation middleware, the run-time of a MSA based application will consist of a set of ICE objects, on each of the client and the server end-points. On each end-point these objects form a hierarchy which closely follows the structure of a FIDL++ declaration. The same structure have been encountered in Section 8.2, which presents the class names generated by our FIDL++ compiler.

Before we consider all these individual objects, we present an overview of the main components of a MSA based application.

9.1 Main components

Figure 9.1 shows the main components of a MSA based application. If we compare this figure to Figure 3.3, which shows the structure of a regular ICE based application, we see that in a MSA based application the run-time environments are symmetric on the client and the server side of the application, as opposed to a regular ICE (and CORBA) based application. We also see that *all* components of both sides of a regular ICE application are present on both the client and the server side of a MSA based application, i.e. both *proxies* and *skeletons* are employed on both sides of the application.

The MSA, shown as the component on top of all other components, is interfacing all of them. It uses both proxies and skeletons to implement its functionality, and it needs to occasionally interface the general ICE API, for purposes like reading configuration files, and the ICE's object adapter in order to register it's own object implementations (based on skeletons) with it.

The dotted, vertical arrows, suggest the fact that in a MSA base application the client and the server applications can choose to combine the

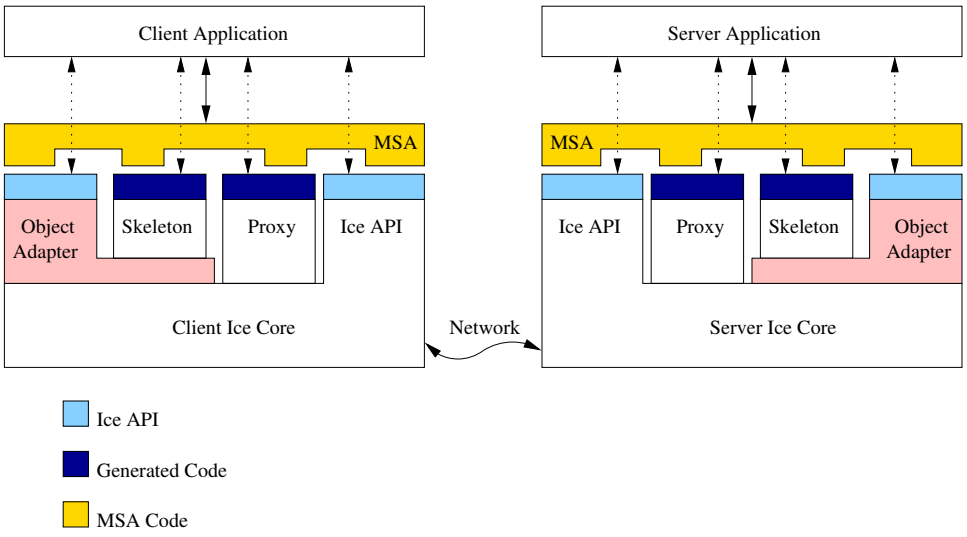

Figure 9.1: *MSA based application structure.*

use of the MSA with directly interfacing the regular ICE components of an application.

9.2 ICE based server callback implementation

In Section 5.2.2 we have presented the principles of the server callback programming approach, for a general ORB. Since we use this approach in the implementation of our MSA, we present here the specific implementation of this approach for ICE.

Figure 5.5 shows the general initialization routine for a server callback based application. In Figure 9.2 we show the objects which are involved in the establishment of a flow of data in an ICE based server callback application.

The dotted vertical line represents the separation of the client's and the server's address spaces. After the object adapters (not shown) have been created and initialized on both the client and the server application sides, we create a smart pointer handle and a proxy handles for the class involved. These are depicted by the colored objects.

Somehow, a proxy to s smart pointer on the server must be given to the client. As with CORBA, we often use stringified representation of remote objects. In our application, we read the server's smart pointer stringified representation from the client's configuration file.

The initialization of the server callback communication pattern bestows in the sending of a client proxy to the server, so that the server will be

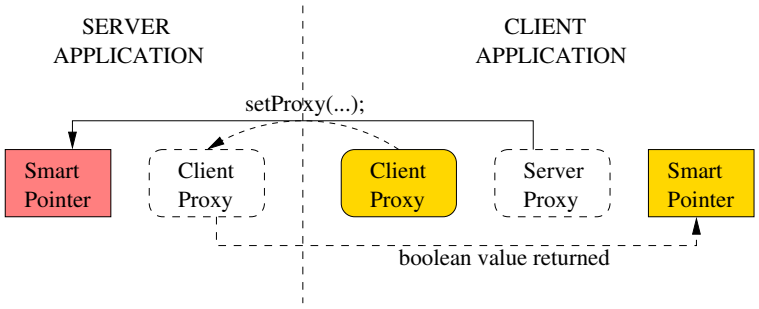

Figure 9.2: *ICE based server callback flow.*

able to make callbacks to the client in the future. The whole initialization routine centers around the use of the special purpose setProxy operation, which is defined by our compiler for all the classes (derived from a FIDL++ specification) which need it. Its definition is:

```
void setProxy(...);
```

We have defined it to return a void, i.e. nothing, but it should be defined to return a boolean value if it is important for the client to know that the server has received the client's proxy. This is represented by the dotted arrow at the bottom of the figure.

The parameter taken by this operation is a proxy to the class for which the flow will be established. We can say that this operation is a *binding* operation. The arched dotted arrow, which is a tangent to the arrow representing the setProxy operation, depicts the fact that the proxy to one of the client's smart pointer objects is sent as parameter with the operation. The server is waiting for this proxy, and when it receives it, it stores it in a local variable. Figure 9.3 shows the situation after the initialization routine has successfully finished, and the flow between the client and the sever sides of the application is therefore established. We see how the client and the server sides of the applications can now exchange both control and payload data.

9.3 Compiler generated objects

There are three features of a FIDL++ specification which find a direct representation in the MSA run-time:

1. The media elements
2. The interpretation we have given to the constraint clause
3. The stream object

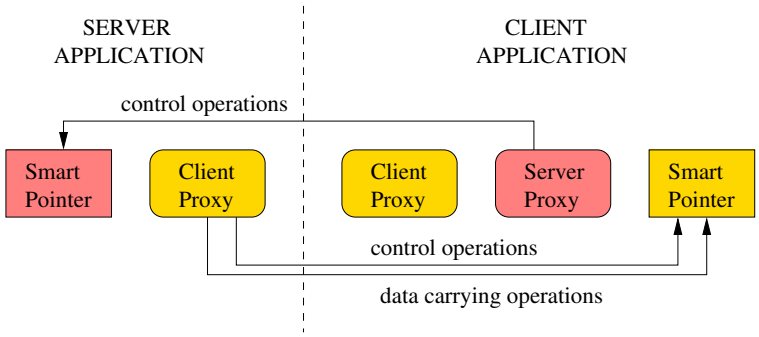

Figure 9.3: *ICE based server callback flow.*

For both of these two features we create objects in our application, and we "pair them up" according to the server callback communications mechanism presented in this section.

9.3.1 Media elements derived objects

Since each media element present in a FIDL++ declaration can be part of at least one flow implementation, we create and initialize objects for each of them, in both the client and the server side of the application. For *every* element declaration, we generate a sets of objects as the ones shown in Figure 9.3. Figure 9.4 shows a scenario in which n elements have been declared.

These objects are created from the class declarations described in Section 8.2. These objects are also the objects which interface the MEMs, described in Section 8.3, and represented by the black dots in Figure 8.1.

9.3.2 Constraint clause derived objects

The interpretation we have given to the constraint clause tells us how many flow implementations are possible for the FIDL++ specification being parsed. For each possible flow implementation, we create 2 kinds of objects.

Flow implementation objects

Each flow implementation object provides a logical encapsulation of all those element objects which it is made of. Figure 9.5 shows a scenario where we have m possible flow implementations.

In this figure we show that the first set of these new objects represent a logical encapsulation of the first and the second element declarations, because each object in this set encapsulate the correspondent objects in the sets for the first and the second element declarations. With a total of n element

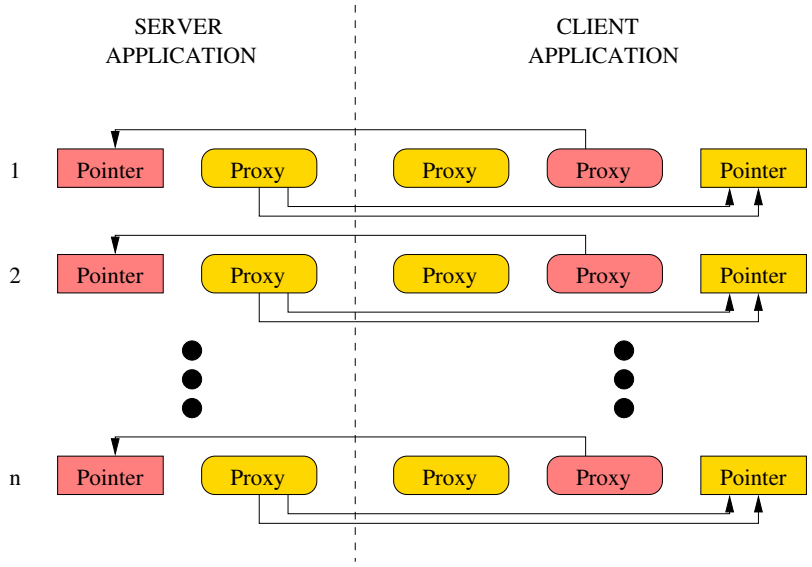

Figure 9.4: *Objects corresponding to element declarations.*

descriptions, a flow implementation object can encapsulate any number or element objects, from 1 to n, not only 2 as shown in this example.

What is new for the flow implementation objects is that we do not use them for the transfer of payload data, but only for control management. This is represented by the fact that one of the arrows representing operations from the server to the client is dotted. It turns out that for the purposes we have encountered so far, it is sufficient to transfer data at the level of objects representing the element declarations, but also this objects can be made to transfer data. Maybe this would be useful in a scenario where we want to transfer data at a larger granularity than what the element description's.

These objects are created from the class declarations described in Section 8.2.

QoS level objects

For each possible flow implementation we create one *binding* object. The purpose is to have a single point of control for all those objects belonging to the flow implementation which interface the MEMs. QoS level objects are created only on the client side of the application. In Figure 9.6 they are represented by the blue squares.

The two objects which are encapsulate by the QoS level objects are all what is needed to control the objects which interface the MEMs. The golden yellow component of each QoS level object (CFI Ptr) controls the MEMs on the client side directly, as shown in Figure 9.5. The red component of the

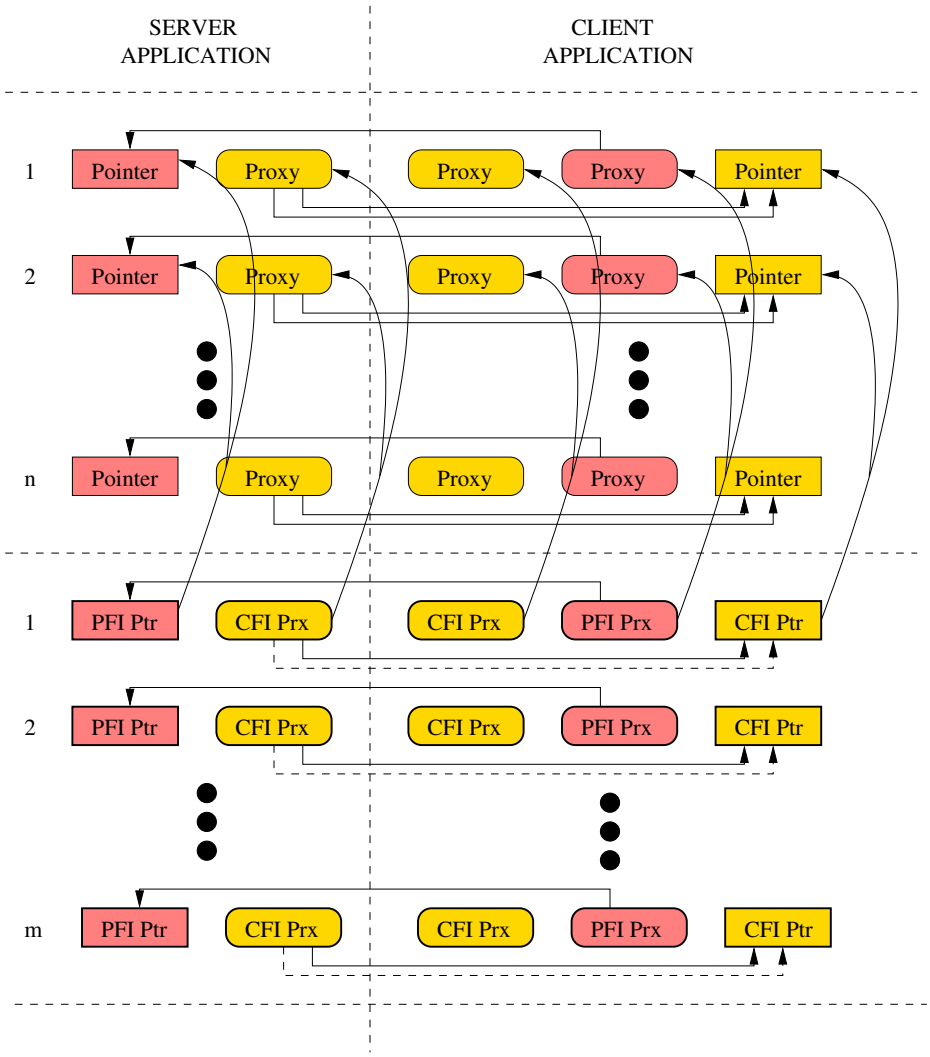

Figure 9.5: *Objects corresponding to flow implementations declarations.*

each QoS level object (PFI Prx) can be used by the client to invoke control operations on its corresponding object on the server (PFI Ptr). This object controls the MEMs on the server side, as shown in Figure 9.5.

The QoS level objects are the closest we come to the RM-ODP concept of *binding* in our implementation of the MSA.

These objects are created from the class declarations described in Section 8.2.

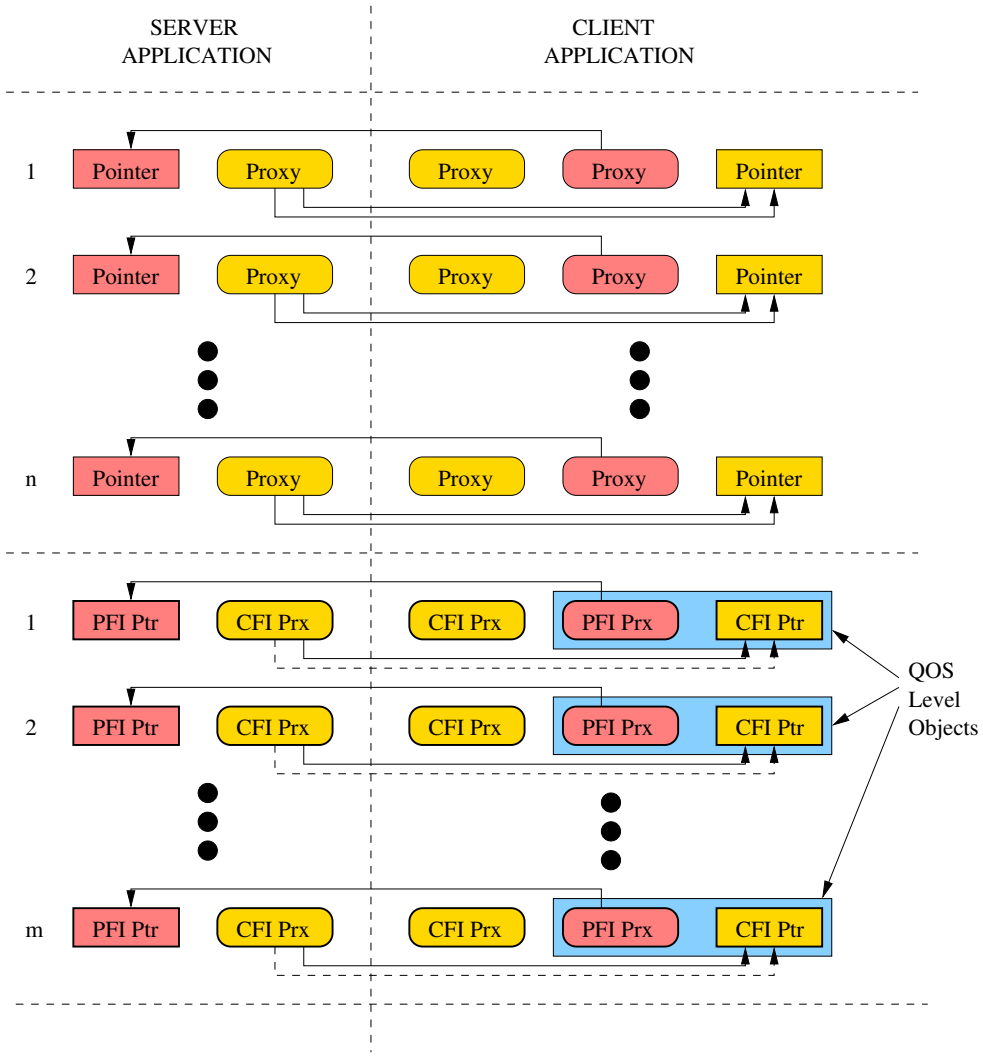

Figure 9.6: *QoS level objects of the MSA's run-time.*

9.3.3 The stream object

Finally, we have arrived at the stream object, which is at the top of our hierarchy of objects. It is represented in Figure 9.7 by the dark blue square around the QoS level objects.

This object is an instance of the class described in Section 8.2, and it is the incarnation of a FIDL++ specification. In our design, all streaming API is implemented by this object.

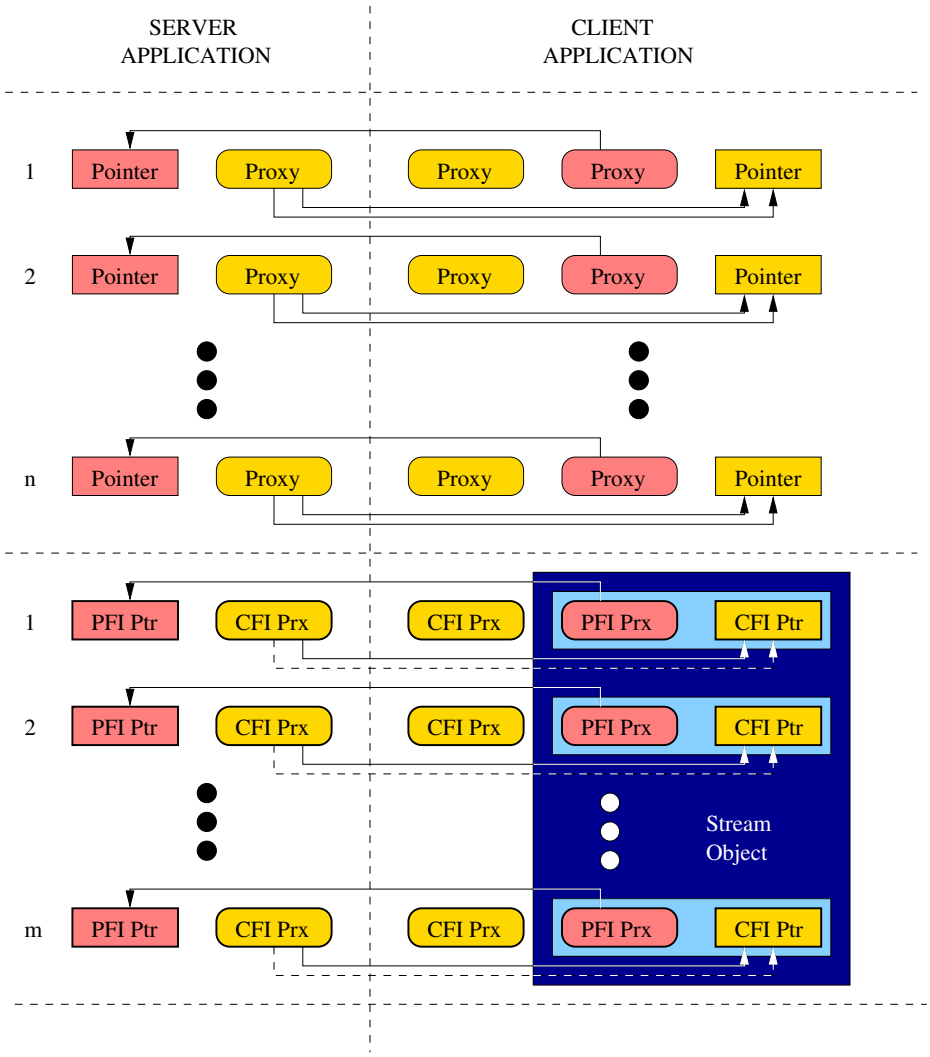

Figure 9.7: *The stream object of the MSA's run-time.*

9.4 Summary

In this chapter we have presented all the objects which make up the run-time of a MSA based application. In addition, Section 9.2 has described how the general server callback programming mechanism can be implemented with ICE.

In the next chapter we present the streaming API provided by our MSA, which consists of the methods which can be invoked on the stream object at which we have arrived in this chapter.

Chapter 10

The streaming API

Our overall goal, in this thesis, is to be able to perform at least such basic operations on a stream object as stream, stop and pause. As we will see in this chapter, our API provides a few other QoS related operations as well.

10.1 Basic operations

The stream, stop and pause operations are mappings of well established functionality of media players, such as walk-mans, CD-players, mini-disk players, MP3 players or a VCR. We have not provided any recording functionality by means of our API, but we argue that once the networking is functional, it is not so difficult to save to disk, in some appropriate format, the data which is streamed.

The meaning of the stream operation is that of *starting* or *playing* a stream. We will therefore have to differentiate from the context between *stream* as a verb, implying an operation on an object, and *stream* as a noun, denoting a stream object.

Specifically, if we have a stream object called A in an application, we want to be able to start, stop and pause the stream object by means of the following statements:

```
A.stream();
A.stop();
A.pause();
```

Because a stream object is an encapsulation of other objects, we need to look closer at the meaning of each of these basic operations.

A stream object can be incarnated by several flow implementations, as dictated by the interpretation we have given to the constraint clause. When a stream specification is parsed, the compiler will generate a class declaration for each possible flow implementation. When a stream object is initialized, it will contain pointers to flow implementation objects for all possible flow

implementations. There will always be only one object for each possible flow implementation.

Since all flow implementations are mutually exclusive, only one of them can be ***active*** at any time. The fact that a flow implementation is active must not be mistaken with a stream object's Streaming state. The fact that a flow implementation is active means only that it is *that* flow implementation to which the stream object will forward its requests when the stream object's methods are invoked. Those flow implementation objects which are not active must necessarily be ***inactive***. Thus, a stream object can be in any of its states (Streaming, Stopped, or Paused)[1], regardless of which of its flow implementation objects is active. Also, a stream object will always have an active flow implementation object, even if the stream object is Stopped.

A stream object keeps track internally of which flow implementation object is the active one, and always forwards any of these basic operations to the right object. We will see in the next section how to programmatically choose which flow implementation object to be the active one.

10.2 setQOSLevel

As we have mentioned in Section 7.8, we have made the concept of QoS level a synonym for a flow implementation. We have therefore declared and defined a method called setQOSLevel for the stream object, which allows us to instruct the stream object at which QoS level of data to stream, i.e. which flow implementation to make active. The method has the following Slice declaration, and it is completely independent of the contents of the parsed FIDL specification:

```
bool setQOSLevel(int qosl);
```

The integer argument to this call must be a value between 1 and the number of available flow implementations. For a stream object called A, the following statement will request the most preferred QoS level:

```
A.setQOSLevel(1);
```

This method returns a boolean value, specifying if the stream object has succeeded in (re)configuring itself to stream at the requested QoS level. Upon failure, the previous QoS level will remain in use.

In the case of our demo application, requesting QoS level no. 1 means that we want VGA sized images and Sound. However, simply requesting QoS level no. 1 does not mean that we also automatically have requested data with the *best possible* QoS characteristics. Recall the FIDL specification of the VGA and the Sound media elements:

```
image VGA {
```

[1] A stream object should never be in an Undefined state.

```
    producerElement = "EWebCam";
    consumerElement = "EImageViewer";
    encoding = "YUVp420";
    samplerate = (5, 15);
    height = 480;
    width = 640;
  };

  audio Sound {
    producerElement = "EMic";
    consumerElement = "ESoundCard";
    encoding = 2;
    channels = {1, 2};
    samplerate = {8000, 11025, 22050, 44100};
    samplesize = {8, 16};
  };
```

Because the attribute assignments of the media element specifications are parsed from left to right, all attributes will have as their default value the least value they can have. For instance, the frame rate of the video, given by the samplerate attribute, will have 5 as its default value, but 15 as the value which gives *the best* QoS data within this QoS level. In the same way, the default sound of this most preferred QoS level is mono, 8 bits/sample, and 8000 samples per second.

We must therefore differentiate between the *default QoS sublevel* and the *best QoS sublevel* within any given QoS level. Whenever we reset the stream object to a new QoS level, it will return to the default sublevel of the requested QoS level. The best QoS sublevel is defined to be that sublevel in which *all* attributes of *all* constituting media elements have their *best* value. The best value means ususally the largest value, but if we should define attributes like latency it would obviously mean the least value for these attributes.

We can manipulate the FIDL specification so that the best sublevel of a QoS level is also the default sublevel. We can do this by specifying the greatest values first, because FIDL does not require that the lowest value an attribute can have must be specified first. Thus if we had specified the VGA and the Sound media elements as in the following example, the best QoS sublevel and the default QoS sublevel of QOSL1 would coincide:

```
image VGA {
  producerElement = "EWebCam";
  consumerElement = "EImageViewer";
  encoding = "YUVp420";
  samplerate = (15, 5);
  height = 480;
  width = 640;
};
```

```
audio Sound {
  producerElement = "EMic";
  consumerElement = "ESoundCard";
  encoding = 2;
  channels = {2, 1};
  samplerate = {44100, 22050, 11025, 8000};
  samplesize = {16, 8};
};
```

While this is totally possible, the compiler must be implemented in such a way as to be aware that a range value can be specified in descending order too. All other FIDL value types are naturally commutative, because FIDL sets are unordered, and all value types are reduced to sets during the parsing process. We will dwell on this issue in more depth in the chapter about further work.

In order to choose between the available sublevels of any given QoS level, we have further defined other API calls, which we present in the next section.

10.3 Setting individual attributes

One QoS sublevel of a given QoS level is differentiated from another QoS sublevel by the fact that at least one attribute of one media element has a different value. Because it has proven difficult to find a good naming scheme so that we could define types for each sublevel of a QoSlevel, we have decided to provide an indirect means of setting the desired QoS sublevel, by means of methods which set the value of a single attribute.

Let us consider our demo application again. Looking at its specification in Program Listing 7.1, we see that every media element declared there has at least one variable attribute. Since the names of the media elements are unique, it is tempting to settle for a naming scheme which uses the media element's name as the basis for the method names. For instance, for setting the framerate of the VGA element (an attribute of type integer), we could be tempted to define a method like bool setVGAsamplerate(int samplerate) However, while this would be easily accomplished in the compiler, it would probably not be so usefull as it appears to be.

One crucial characteristic of streaming with the MSA is that it is easy and straight forward to change the QoS level of the stream object. As an example, if the stream object of our demo application is operating at a QoS level which does not make use of the VGA media element (QOSL2 or QOSL3), the call to setVGAsamplerate, while legal, would not accomplish anything. The programmer would then have to remember which elements belong to which QoS level and make sure that the right call is made. This can lead to quit lengthy if tests (or the equivalent) if there are many possible flow implementations.

Since, ultimately and regardless of the employed naming scheme, the call to set an attribute will be dispatched to a module which implements the functionality of the media element, we thought that maybe it would be easier if we provided API names based on the names of the functionality implementing modules.

Each media element declaration must specify which modules are implementing its consumer and producer side functionality. For this purpose we use the the producerElement and the consumerElement attribute names which we have added to FIDL, as mentioned in Section 7.5.

Thus, for each media element declaration which has at least one attribute whose value can be set, we will have to interface two modules. Strictly speaking, we do not require that the producer and the consumer modules must be different, but we so far we have not conceived any situation where it would be useful to have media elements which have the same producer and consumer module.

Also, as we can see from the FIDL specification of our demo application, a module can implement the functionality of several media element declarations. All three of the VGA, the SIF and the QSIF elements will be served by the same webcam module and all three of them will serve the same image viewer module.

All attributes which have values which are not *atomic* in the FIDL specification can be set. While parsing the FIDL specification, our compiler will pick up these attributes and declare methods for them, based on the modules these attributes ultimately point to. The names are made up by the set prefix, followed by the name of the module and followed by the name of the attribute.

It is possible to employ this naming scheme, because the stream object always dispatches the calls made to it to the active flow implementation object.

As an example, in order to set the value of the samplerate attribute of any of the VGA, SIF or QSIF media elements, we have to make calls to the setEWebCamsamplerate and setEImageViewersamplerate methods. The first one is for the producer module and the second one is for the consumer module, which are the same for all of these media elements.

We consider that it is easier for a programmer to think in terms like *"in this situation I need to set down the samplerate of the webcam"* rather then *"in this situation I need to know which flow implementation object is active and set down the samplerate of that media element which interfaces the webcam module"*.

The methods defined and declared for setting attributes are highly dependent on the parsed FIDL specification, so we could not give a more concise and generalized presentation of this subject. We will conclude this section by listing here the methods which have been declared for our demo application in the interfaceImplementations_callback.h file:

```
bool setEWebCamsamplerate(int samplerate);
bool setEImageViewersamplerate(int samplerate);
bool setEMicchannels(int channels);
bool setESoundCardchannels(int channels);
bool setEMicsamplesize(int samplesize);
bool setESoundCardsamplesize(int samplesize);
bool setEMicsamplerate(int samplerate);
bool setESoundCardsamplerate(int samplerate);
```

10.4 Extra API calls

We have implemented one more kind of API calls, because it seemed to be very necessary to begin with. Eventually it proved to be somewhat of an unhealthy approach, so we present it here as a case study only.

In the previous section we have described methods used to set a media elements attributes. While we worked on it, we were tempted to also provide API calls for the basic operations at the producer and consumer module level. In other words we wanted to be able to say to a specific producer or consumer module which implements the producer or consumer functionality of at least one media element, that we want it to start, stop or pause. This seemed to be a desirable feature because it proved to be such a good approach to provide attribute setting API calls at the same level.

However, we discovered that, while it is not so difficult to provide these API calls, using them sets a stream object in the Undefined state.

As an example consider the following API calls which our compiler actually does generate code for:

```
void EWebCamstream();
void EWebCamstop();
void EWebCampause();

void EImageViewerstream();
void EImageViewerstop();
void EImageViewerpause();

void EMicstream();
void EMicstop();
void EMicpause();

void ESoundCardstream();
void ESoundCardstop();
void ESoundCardpause();
```

Let's say that our demo application is running at any one of its three QoS levels and that the stream object is in its Streaming state. Since a stream object always dispatches any calls to the right objects, calling any of the above mentioned functions will be send to the objects of the active

flow implementation. If the call was a call to stream, nothing will happen, as the streaming object is already streaming, except that some network resources will be wasted. If the call was made to say EMicstop(), something very interesting will happen: the module which implements the microphone functionality will be stopped. This brings the stream object in the Undefined state, because now some of the modules which work for the currently active flow implementation are still streaming while one of them has stopped. In other words the stream object is partially Streaming and partially Stopped.

This can actually be a desirable feature. The example we have given is that of muting the microphone during a webcam chat session. This is useful, and has been implemented in many regular phones and in many multimedia applications similar to our demo. What is incorrect in this example is not the muting of the microphone in itself, but the fact that it is done in a way which brings the stream object in the Undefined state.

The sound way of doing this would be to rather reconsider the constraint clause of our demo application's FIDL specification. If we change it from what it is now:

```
constraint (VGA | SIF | QSIF) & Sound;
```

to

```
constraint (VGA | SIF | QSIF) & Sound
           | VGA | SIF | QSIF | Sound;
```

our compiler will interpret it as follows:

```
Constraint interpretation:
--------------------------
1: ['VGA', 'Sound']
2: ['SIF', 'Sound']
3: ['QSIF', 'Sound']
4: ['VGA']
5: ['SIF']
6: ['QSIF']
7: ['Sound']
```

The difference is now the fact that there are four more legal flow implementations, or QoS levels, which by virtue of being constituted of a single media element each, provide us with the possibility to stream any size of images without sound, or sound without any images. Instead of using such unhealthy calls as we have exemplified in this discussion, we can now mute the microphone by setting the QoS level of the stream object to any value between 4 and 6, by calling setQOSLevel. This will achieve our goal and still leave the stream object in a sound Streaming state.

10.5 Summary

In this chapter we have provided a complete overview of the streaming API provided by our MSA. We also presented a case study which shows how powerful the constraint clause of a FIDL++ specification can be.

In the next chapter we present the demo applications implemented to prove the versatility of our design and implementation of both Da CaPo and the MSA. In our main demo application we test all features of the API presented in this chapter.

Chapter 11

Demo applications

While working with this thesis we have made implementations of many demo applications, in order to convince ourselves of the soundness of our design. We present here three of them. They all are streaming images from a webcam device handled by one process, to an image viewing application handled by another process. What is different between these applications is *how* the images are streamed from one process to the other.

11.1 Da CaPo demo

In order to demonstrate our implementation of a Da CaPo core, we have implemented a few modules. These are located in the base directory, and their files names begin with the m_ prefix.

Figure 11.1: *Da CaPo demo: surveillance camera.*

Our Da CaPo demo is a simple surveillance camera application, running over TCP. Figure 11.1 shows how the application looks like when it is run. The main window of this application, shown to the right in Figure 11.1, shows up only because it is coded as part of image viewer module which we

Producer side	Consumer side
MWebCam	MImageViewer
MTCPSender	MTCPReceiver

Table 11.1: Modules used in our Da CaPo demo.

have used, but it's controls are not linked to any functionality. The purpose of this demo is to show how to build protocol graphs with our Da CaPo core implementation, and because of lack of time, we have not implemented any advanced functionality in this application.

In this demo we do not make use of any C-modules, because we do not need any intermediary networking functions. We link the producer and the consumer modules (A-modules) directly to the sending and receiving modules (T-modules), and transport the uncompressed data provided by a webcam, over TCP, to an image viewing application which also is able to display uncompressed images. Figure 11.2 shows the protocol graph of our demo. If we compare it with Figure 6.3, we see that it is only a more specified case of the general scenario presented in Section 6.1.2.

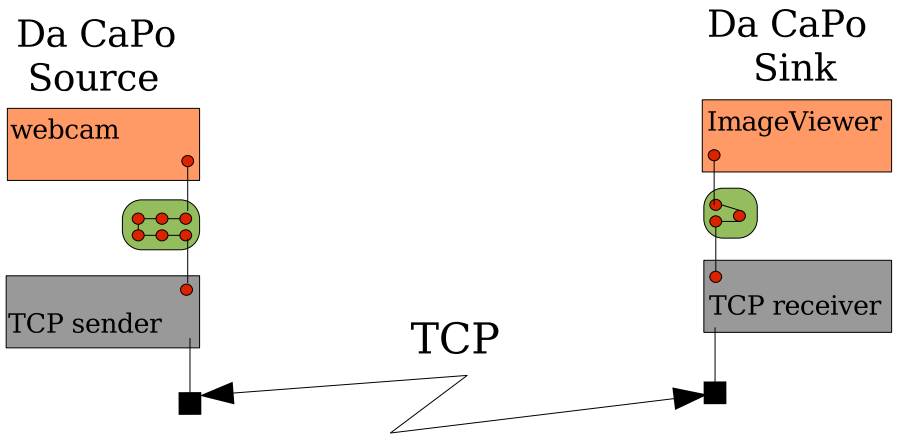

Figure 11.2: *Da CaPo demo protocol graph*

Table 11.1 shows which classes implement the modules depicted graphically in Figure 11.2.

What is of particular interest here, is how simple it is to link modules together in module graphs, by means of only the constructor of the Modulel class.

Following the algorithm presented in Section 6.3, the following code shows how to implement the module graph on the producer side of our demo application:

```
queue<DCPacket *> buffer;
```

```
IceUtil::Mutex mutex;

IceUtil::ThreadPtr mWebCam =
    new MWebCam( ..., 0, &buffer,
                      0, &mutex, ...);

IceUtil::ThreadPtr mTCPSender =
    new MTCPSender(..., &buffer, 0,
                        &mutex, 0, ...);
```

What we see, is that pointers to the buffer and the mutex variables are given to the constructors of the modules, in such a way that buffer becomes output buffer for the webcam module and input buffer for the TCP-Sender module. In the same way, mutex become output buffer mutex for the webcam module and input buffer mutex for the TCP-Sender module.

In the same way, the following code shows how to implement the module graph on the consumer side of the application:

```
queue<DCPacket *> buffer;
IceUtil::Mutex mutex;

IceUtil::ThreadPtr mImageViewer =
    new MImageViewer( ..., &buffer, 0,
                           &mutex, 0, ...);

IceUtil::ThreadPtr mTCPReceiver =
    new MTCPReceiver(..., 0, &buffer,
                          0, &mutex, ...);
```

On the consumer side the buffer plays the role of output buffer for the TCP-Receiver module and of input buffer for the image viewer module.

The code for this application is placed in the sender.h, sender.cpp, receiver.h and the receiver.cpp files. The application is started with the following two commands, and we start the receiver first:

```
[... base]$ ./receiver tcp 50000
[... base]$ ./sender tcp 50000
```

Each of the applications takes two parameters: tcp is compulsory and specifies that this is the protocol we intend to use. For now, it is the only protocol available, but it can be used later to choose something else (like UDP or SSL), when working modules for these protocols will be available. The second parameter must be a legal Linux port number, and it must have the same value for both applications.

This demo is compiled when make is execute in the base directory of our source code. Alternatively, you can call make for these targets only:

```
[... base]$ make sender receiver
```

11.2 ICE based polling client demo

This demo application is an ICE based version of the Da CaPo demo. It uses the polling client programming paradigm, which has been presented in Section 5.2.2.

This demo has the same appearance as the Da CaPo demo, but it's functionality is more complete. Figure 11.3 shows how we can vary both the image size and the framerate of the surveillance camera.

The purpose of this demo is to show how we can implement a form of streaming using only the "simple" abstractions offered by a RPC oriented platform for distribute computing, like ICE.

In this application, we declare operations for both control and data transport purposes. The client is the active entity and it polls the server for each single image. The Slice declarations are straight forward:

```
module dacapoP {
  sequence<byte> DCPacket;
  enum ImageFormat {YUVp420};

  interface iWebCam {
    DCPacket getImage(int width, int height,
        ImageFormat if);
    int setImageSize(int width, int height);
    int setFrameRate(int frameRate);
  };
};
```

The only unexpected feature could be how the framerate is regulated. If the client is the active entity, why should we need a setFrameRate operation? After all, the client can poll for images as often as it desires, and depending on the networking conditions, it will always get up to the desired number of frames per second. Such functionality can easily be implemented by means of a timer in the client, whose accuracy can be trusted at a granularity of, say 1/100 of a second.

By implementing an operation for setting the framerate, we show that we let the server enforce this parameter. The reason is found in the way our test webcam functions: you can poll it for images as often as you want, it will only deliever them at its current framerate. Using this feature, we let the client poll for a new image at once it has displayed the previous image it has received. If it polls to early, it will have to wait until the webcam serves the next image. This waiting behavior is enforced by ICE, because the getImage operations is a two-way operations, and the caller blocks until the result arrives.

We note also that the image data is transported by means of variables of type sequence<byte>, which is mapped in C++ to a vector<uchar>. byte is the only Slice data type for which ICE guarantees that the data's internal

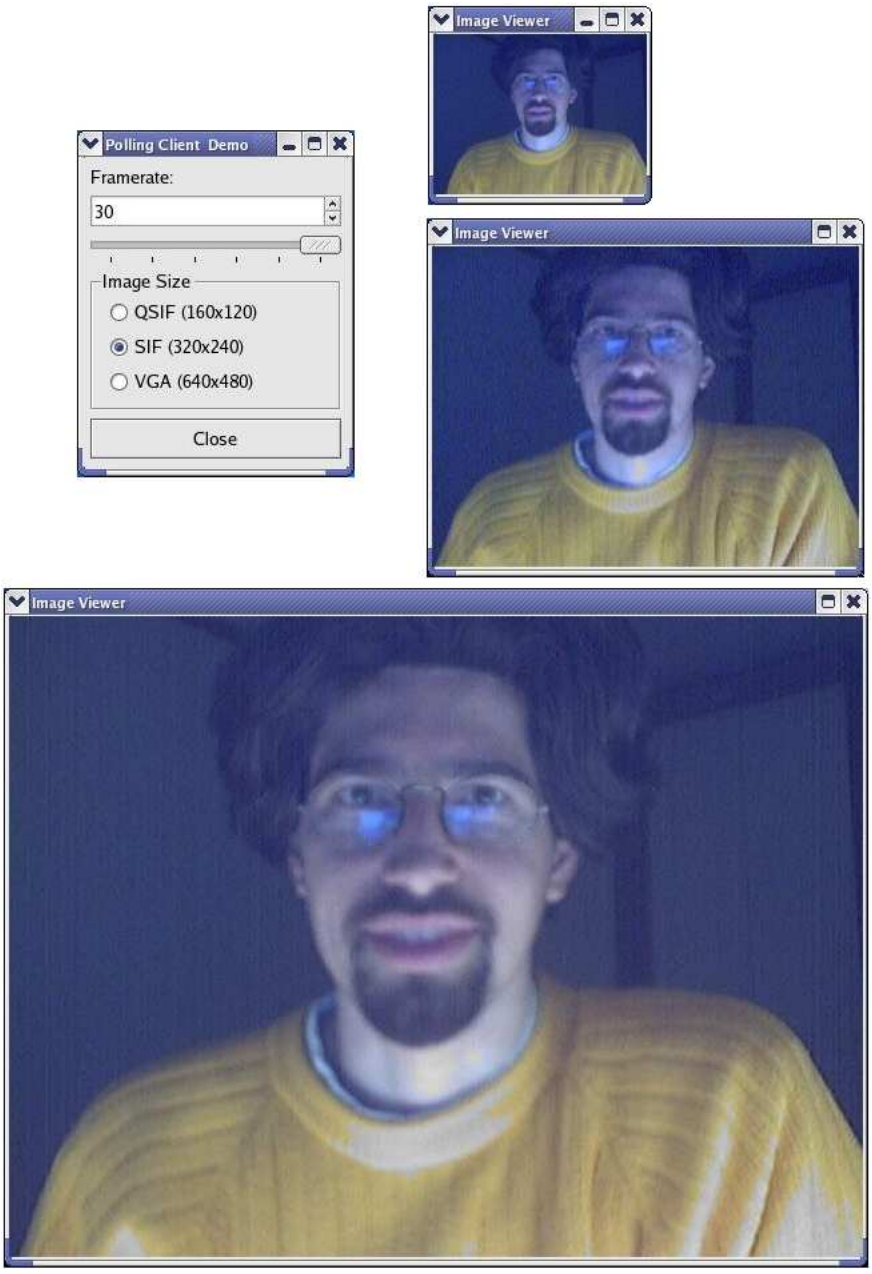

Figure 11.3: *ICE based client polling demo application.*

bitwise representation will not suffer any alteration on the receiving host, regardless of the hardware, operating system and programming language used on the client and the server end-points.

The code of this application is found in the clientP.cpp and serverP.cpp files, in the base directory of our source code. The application is compiled when make is run in this directory. Alternatively, you can call make only for these targets:

```
[... base]$ make serverP clientP
```

The executables are called serverP and clientP, and they expect a parameter which specifies the protocol to be used and the port number. This parameter must be provided in an ICE native format, so we start the applications as follows:

```
[... base]$ ./serverP "tcp -p 50000"
[... base]$ ./clientP "tcp -p 50000"
```

11.3 MSA demo application

The MSA demo is our main demonstration application. It puts to the test all the functionality of our compiler and shows how to use every facet of the streaming API developed in this thesis.

The main ideas of this demo is to stream data for two separate flows at the same time and to show in real time the bandwidth used. Then, we change the QoS parameters of the application by means of our API. We change them while the application is running and we monitor the bandwidth used in order to receive a numerical confirmation of the fact that our API calls have indeed the documented effect.

Because of lack of time, we have implemented only *dummy* audio modules for this demo. The producer audio MEM does generate data, which is streamed over to the the consumer audio MEM by the MSA, but the data is not captured by a microphone, and is not fed to the sound card. However, the amount of dummy data generated by the EMic MEM is just as large as it would be it this MEM would really be capturing from a microphone. The ESoundCard MEM registers the amount of data received, and discards it. Therefore, these dummy audio MEMs are good enough for the purpose of evaluating the feasibility of our solution.

Figure 11.4 shows the main and the statistics windows of this demo, while Figure 11.5 shows the image viewer window at its 3 supported sizes.

The GUI of this application is tailored especially to allow us to test all the API features of our solution. On the horizontal axis of the statistics window we show the time, measured in seconds after the start of the application, and the vertical axis show the throughput, measured in Mbits per second.

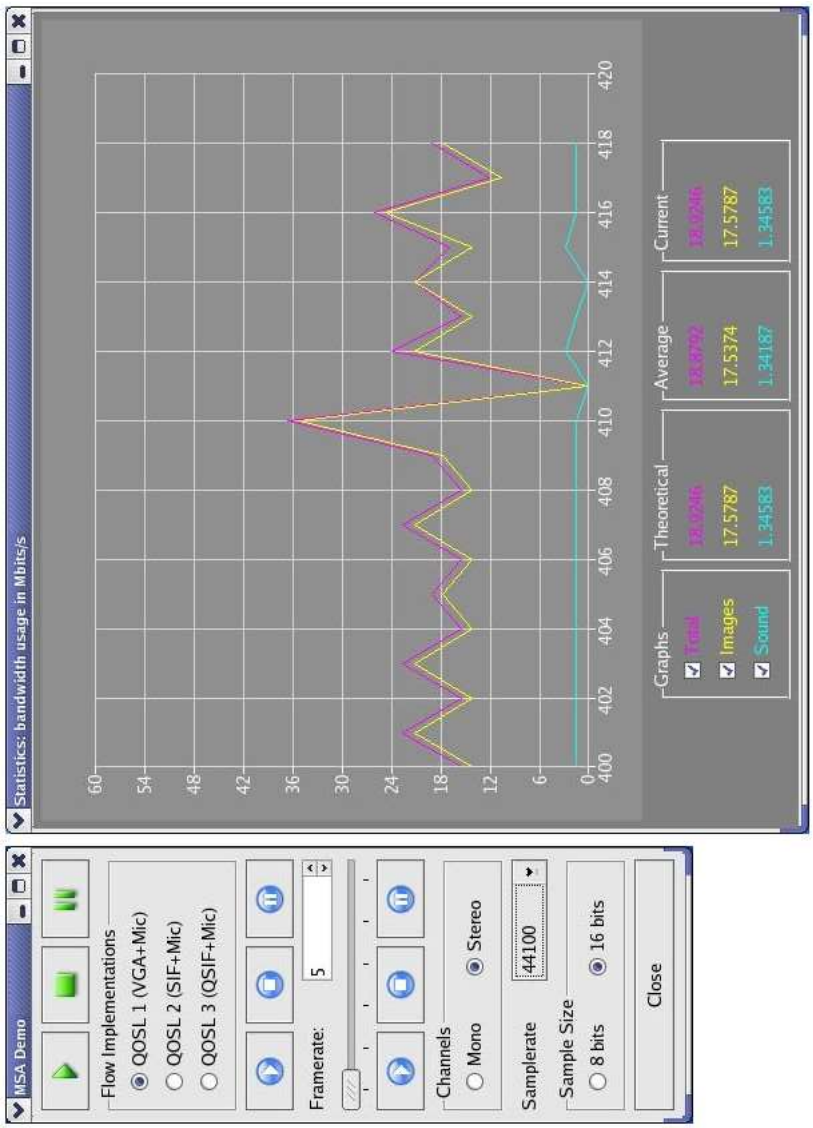

Figure 11.4: *MSA demo: main and statistics windows.*

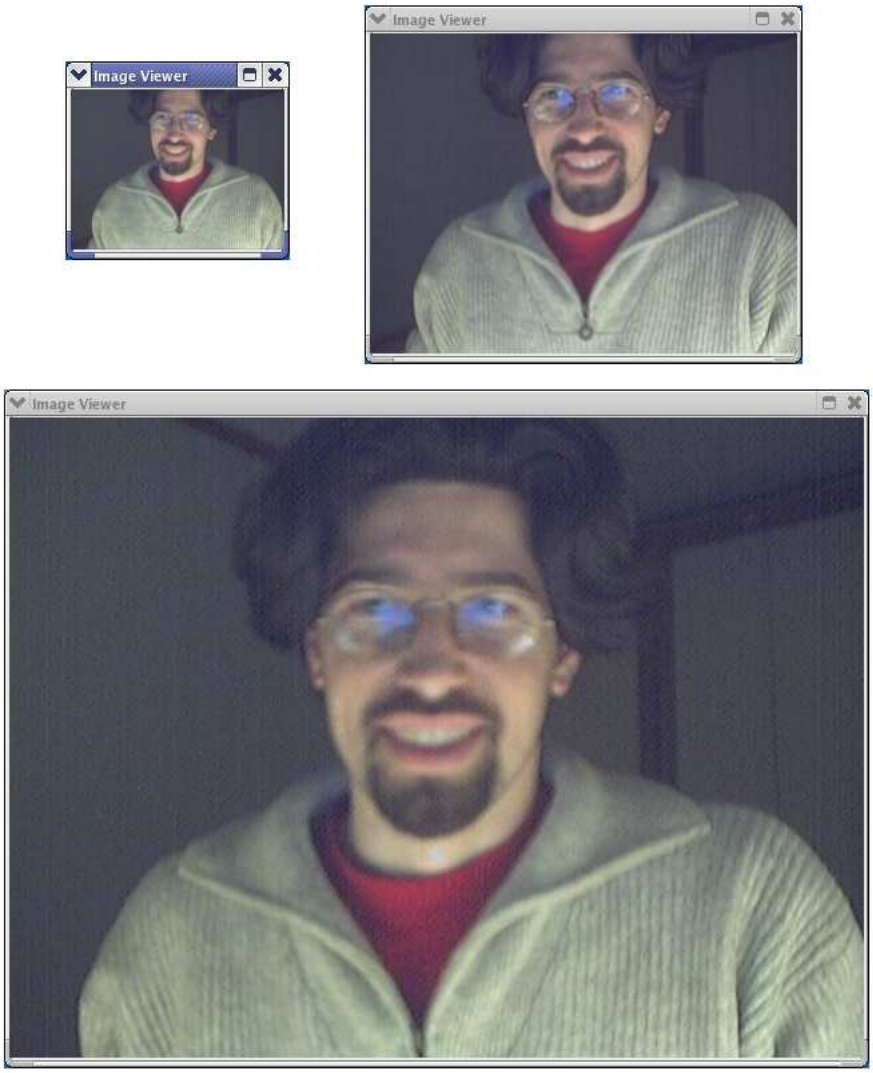

Figure 11.5: *MSA demo: image viewer window.*

Since we stream uncompressed images at up to 30 frames per second the throughput of our application can be quite high.

By multiplying the number of possible combinations of values for the variable attributes of our FIDL++ specification, for each QoS level, and by adding them together afterwards, we see that our application can stream with 976 possible parameter settings. These are too many to give an overview of the theoretical bandwidth requirement of each of them, so we only show here how to calculate it.

The bandwidth required by the image traffic is determined by the size

of the images and by the frame rate. Since we stream images in YUVp420 format, the size of an image, measured in bytes, is 1.5 times its number of pixels. The image traffic, in bits, is then calculated after the following formula:

$$T_i = imageWidth * imageHeight * 1.5 * framerate * 8$$

The size of a second's worth of sound it determined by the values of the how many channels there are (mono, stereo) the sample rate and the size of the samples. A second's worth of sound requires therefore the number of *bits* given by the following formula:

$$T_s = channels * samplerate * samplesize$$

In order to get the total amount of bandwith needed we have to add the two values described above.

We present now a few experiments which we have made, but do not give numbers, except for what can be read from the graphs. The experiments consist in exercising our API by means of the demo's GUI, and looking at the throughput in the statistics window.

- **Green control buttons**.
  The green buttons at the top of the main window are making the API calls described in Section 10.1. They are used to start, stop and pause the stream object. For our MSA, we have defined these operations to mean that all the flows of the stream must be started, stopped or paused when these buttons are pressed. Figure 11.6 shows the effect of pressing the stop button.

  In the graph on the left side, we see that the bandwidth useage has been quite stable over the last 20 seconds. Then we take a snapshot of the statistics window and save it to disk. This extra activity on the machine makes the flow of data, as experienced by our application, more bursty, as shown in the graph on the right side. Still the application keeps the same average values (this can not be seen in the graphs). Then, around second number 115 we click the stop button and take another snapshot as fast as we manage. The graph on the right side shows how the throughputs of both the image and the sound flows drop to 0.

  For this application the pause button has the same effect as the stop button.

- **Flow implementation radio buttons**.
  These three radio buttons correspond to the three possible flow implementations for our demo application, see Section 8.2. Their functionality is to make the API calls described in Section 10.2. Figure 11.7 shows

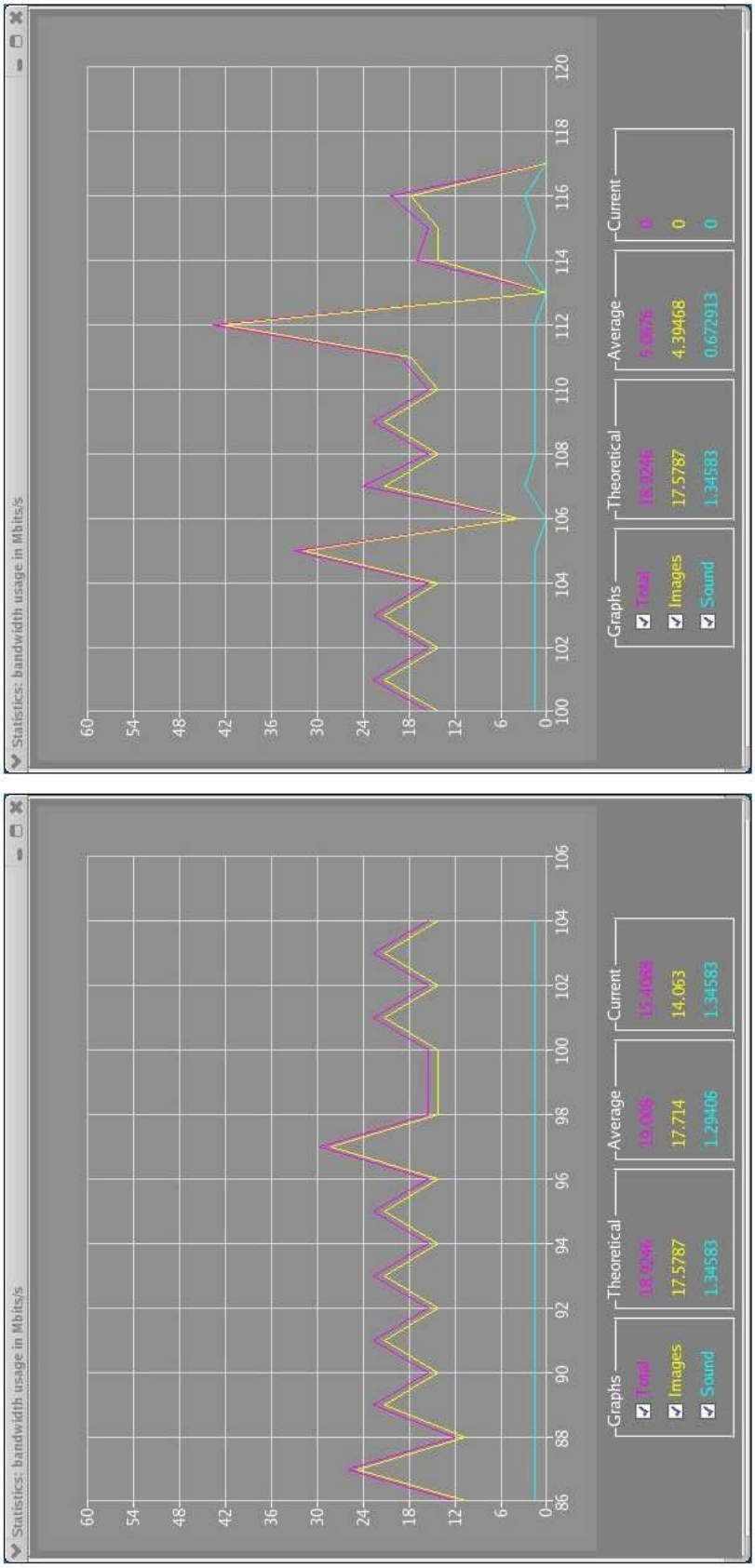

Figure 11.6: *Exp. 1: bringing the stream object to its* **Stopped** *state.*

the effect of moving from QOSL2 to QOSL1, streaming at 5 frames per second, with the best sound in both situations: stereo, 44100 samples per second, 16 bits per sample.

- **Frame rate controls**.
  The frame rate controls are used to regulate the framerate at which the application streams. The slider and the spin box are connected together so they both have the same function. They exercise API calls of the kind described in Section 10.3. Figure 11.8 shows the effect of changing the frame rate from 5 to 30, while streaming SIF images (QOSL2), with the best sound quality at both framerates: stereo, 44100 samples per second, 16 bits per sample.

- **Controls for sound quality**.
  The controls for sound quality exercise the same king of API calls as the frame rate controls, described in the previous experiment. Because the sound traffic is always very small compared to the image traffic, often the graphs for the image traffic and the total traffic will overlap each other. This is exemplified in the experiment shown in graph on the left side of Figure 11.9.

  Here we start with the lowest quality of image traffic and lowest quality of sound traffic. After about 8 seconds we set the frame rate up to the ma, 30, but keep the sound and the image size unchanged. After around 14 seconds we set up the sound to the best quality: from mono, 8000 samples per second, 8 bits per sample (the lowest quality) to stereo, 44100 samples per second, 16 bits per sample. We can see how the graphs for the image traffic and the total traffic split appart, according to the change in the graph of the sound traffic.

- **Blue control buttons**.
  The blue control buttons exercise the API calls of the case study given in Section 10.4. By means of these API calls we can willingly bringing the stream object in the Undefined state. We can see in the graph to the right in Figure 11.9, how the image traffic drops to 0 from time 210, when we stop the image traffic, by means of the stop button in the first row of blue control buttons, and comes back to its normal rate at time 216 after we start it again. The total traffic seems to fall to 0 too, but actually it is exactly equal to the sound traffic, and can therefore not be seen until we start up the image traffic again.

11.4 Summary

In this chapter we have presented the demo applications implemented to prove the versatility of the systems we have designed and implemented in

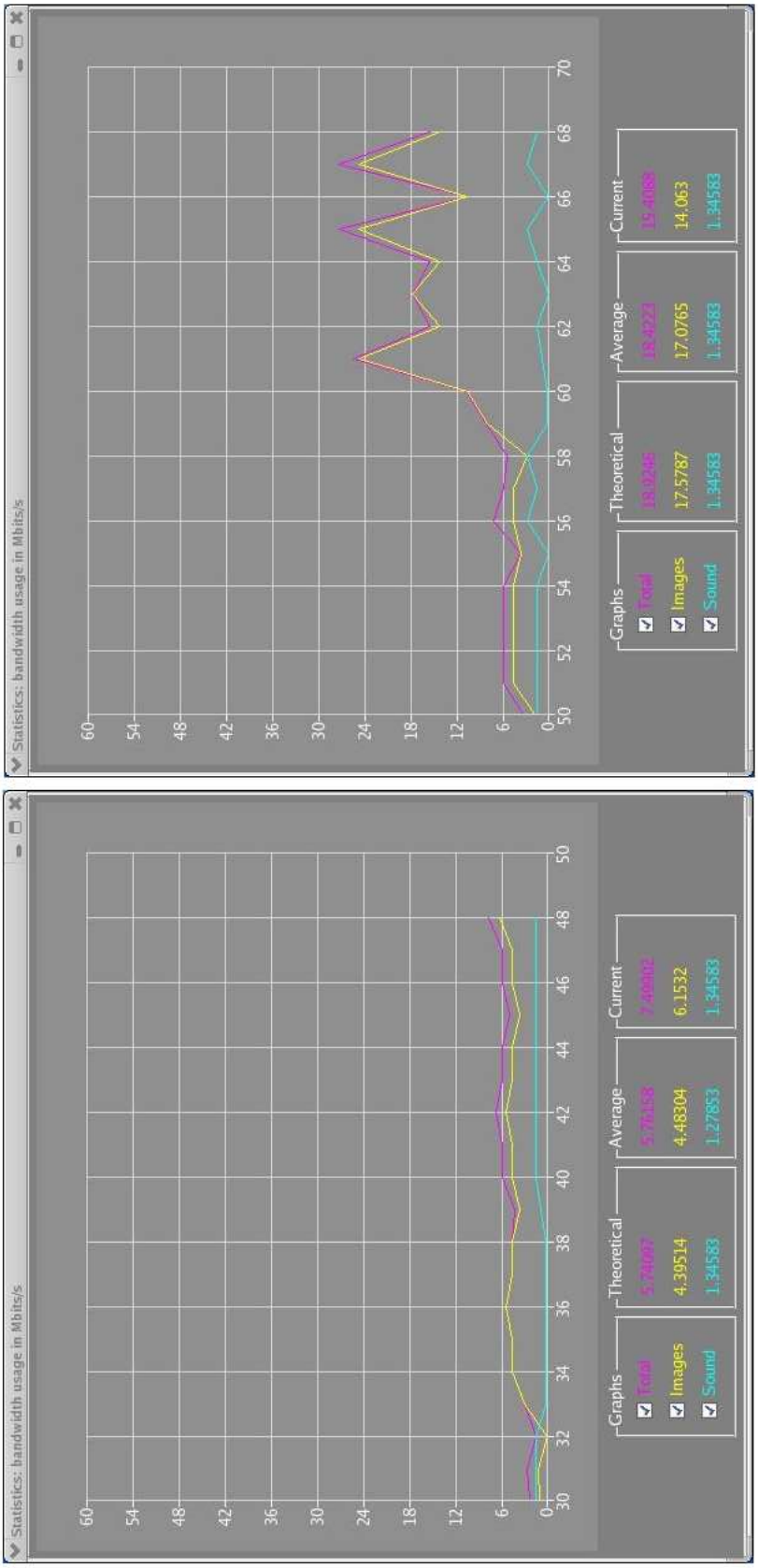

Figure 11.7: *Exp. 2: changing from QOSL2 to QOSL1*

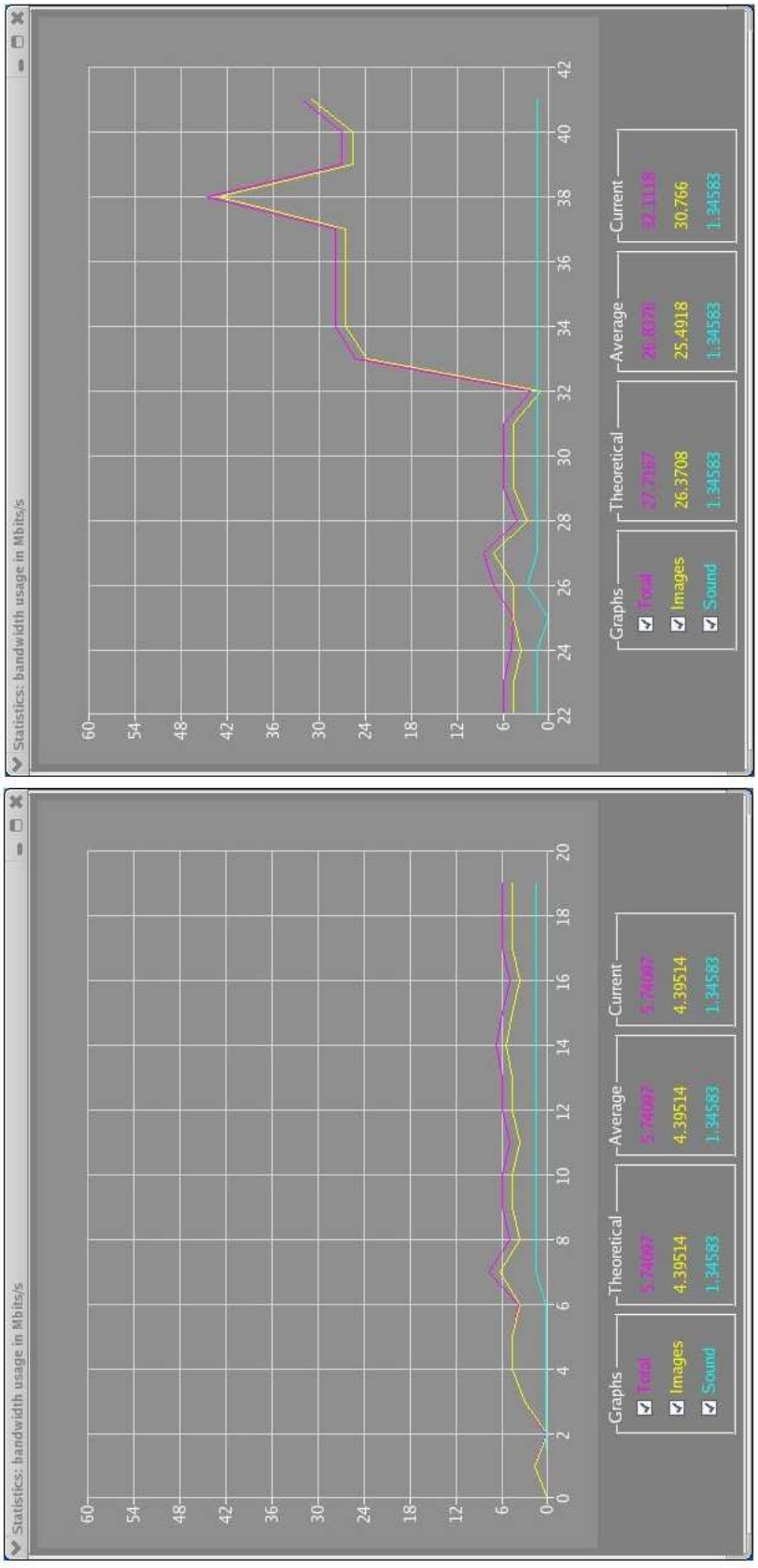

Figure 11.8: *Exp. 3: changing the frame rate from 5 to 30 for SIF images.*

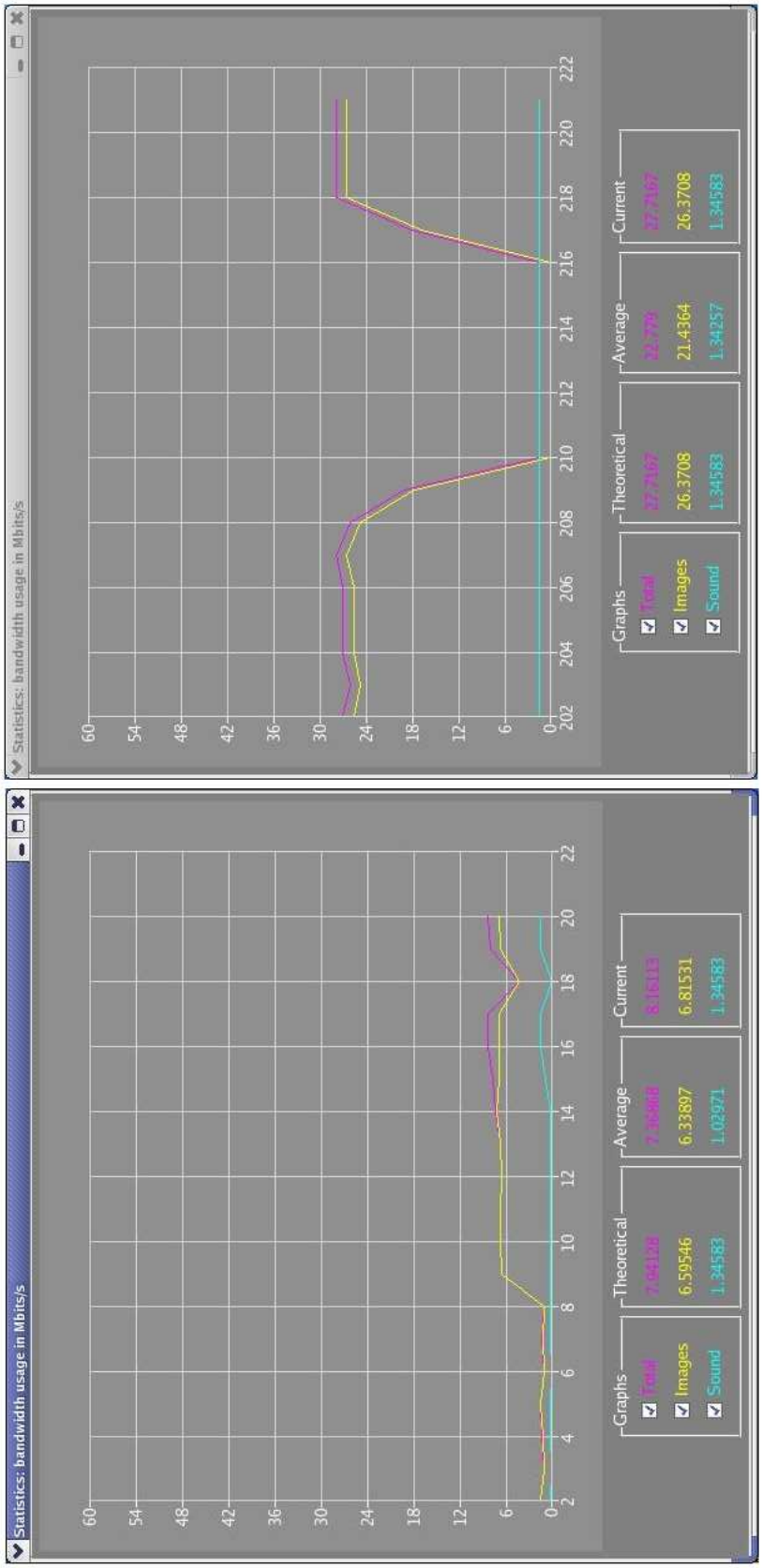

Figure 11.9: *Exp. 4 and 5.*

this thesis.Especially important is the demonstration of the MSA's features. We have presented several experiments which show how the MSA behaves as expected. This brings us towards the end of our work with this thesis.

In the next chapter we present our own evaluation of the thesis.

Chapter 12

Evaluation

In this chapter we present an evaluation of our work. We divide it in two parts: one for our Da CaPo core and one for the MSA.

12.1 Da CaPo core

Da CaPo's strength is its modularity. Modularity leads to flexibility and extensability. These are features which we see also in our implementation of a Da CaPo core.

Because of its extremely simple module interface, it is very easy to build virtually any kind of modules. From the most simple, in keeping in touch with Da CaPo's original philosophy, to as complex as we desire. Simple modules will implement only a simple networking function each. This fine granularity allows the creation of very fine tuned application protocols.

In Section 11.1 we presented our Da CaPo demo application, and we mentioned in Table 11.1 the 4 modules used for its implementation. We have attempted to implement more modules in this thesis, but we did not have time to work on them until they became stable enough.

Among these we have attempted to implement UDP modules. They were an obvious thing to do, because the UDP protocol is very useful for the transport of multimedia data: it is faster then TCP, and an occasional loss of packets is not critical. However, since UDP does not guarantee the delievery of datagrams, we have also implemented *Idle Repeat Request* ***(IRQ)*** modules. Another limitation of the UDP protocol, compared to a streaming protocol like TCP, is the maximum size of a UDP datagram, set to 65.535 bytes [10]. Our experiments with streaming uncompressed images have shown us that it is easy to exceed this limit. The natural thing was then to implement fragmenter and defragmenter modules.

The fragmenter module would check the size of a packet, and if it is larger than a certain threshold value (close to the UDP datagram limit), it would fragment it up and push the fragments in its output buffer, instead of

the original packet. For this purpose, we have defined the FragmentNumber field in the header of the Da CaPo packet, as described in Section 6.2.3. The defragmenter module would have to read the FragmentNumber field of each packet, and reconstruct those packets which have been fragmented. If a packet is not a fragment, the defragmenter will simply push it in its output buffer so that the next module in the module graph can access it as soon as possible.

By combining the functionality of the UDP, the IRQ and the fragmenter / defragmenter modules, we come close to implementing the functionality of the TCP protocol. What is missing is ordered delievery of packets. This function could also be implemented in the defragmenter module, or keeping in touch with the Da CaPo tradition, rather in another module.

Our implementation of these modules is found in the base directory of our source code. We consider that even though these modules have not reached release stability, they still prove the versatility of our approach. We have managed to preserve in our implementation of the Da CaPo core a simplicity of the module interface which makes it easy to implement new modules. We argue that the problems we encounter with these modules are not due to the design of Da CaPo, but rather difficulties connected to pointers and memory management in C++.

We conclude therefore that we have achieved the first three requirements for a Da CaPo core, which were presented in Section 4.1.2, and revisited in Section 4.2.2. They are: 1) a C++ implementation, 2) simple APIand 3) flexibility.

The 4th requirement, adaptability, is in our opinion the most challenging, and the most difficult to achieve. We have not implemented a proof of concept of this feature in our Da CaPo demo. Still, we reason that, given the extremely simple interface between the modules, each module has an output buffer and/or an input buffer, which both are FIFO queues, it is extremely simply to insert and remove modules into and from a module graph. All what is needed is to reassign the pointers to the buffers to point to other buffers. This simplicity of inserting and removing modules from a module graph is due to the fact that the modules are oblivious of their own position in the module graph. They simply pop a packet from their input buffers, if it is not empty, process it and push it to their output buffers. Therefore, modules will not be disturbed by the fact that they suddenly need to pop packets from another buffer, as long as the packets have a good header and contain valid data.

It is our evaluation therefore, that our implementation of a Da CaPo core is suitable for the programming of adaptable applications.

12.2 MSA

The requirements posed to our MSA, are the combined *guidelines* derived from the requirements posed to MULTE-ORB, presented in Sections 4.1.1 and 4.2.1, and the general requirement of belonging to an object-oriented middleware platform, presented in Sections 4.1.3 and 4.2.3. We go through them all again in this section, in order to evaluate our work.

1. **Dynamic QoS support.**
   As it is defined in Section 4.1.1, dynamic QoS support requires that an application should be able to change its QoS specification while running. This is clearly a feature which we have not implemented support for in our MSA. This is due, first of all, to lack of time. The specification of QoS by means of FIDL++ is very static. It must be given at compile time, and the C++ code generated from it is only a reflection of, or a C++ representation of the semantic meaning of the FIDL++ specification. Since the FIDL++ specification is very static in nature, so is our current implementation of the MSA.

   We have not had time to study what is necessary in order for us to be able to dynamically create the kind of objects which the MSA's run-time is composed of. In other words, if we want to change the QoS *specification* (as opposed to the QoS level) of an application at run-time, how can we generate C++ code to reflect the new QoS specification, compile it and add it to the running application? Clearly this can not be done with a language like C++. We consider that support for dynamic QoS requires the use of object factories of some kind.

   Also, a widened functionality for our MSA is required. In the current implementation, the objects which make up the run-time are created and become a part of the run-time when the application is started up. Dynamic QoS support requires a mechanism for registration of new objects and the creation of new bindings. A regular object adapter is an example of a mechanism used to register active object implementations. When a CORBA IDL or Slice specification is parsed, object implementations can be created for all the interfaces (and classes, for ICE) which have been declared. Such object implementation are created at the request of the programmer, and must be registered with the object adapter before invocations can be sent to them. In our implementation, all possible object implementations derived from a FIDL++ specification are created at start up, and registered with the ICE native object adapter. This makes our solution very easy to use, as our compiler does all the job for us, by generating a rather large amount of start up and initialization code, but it also makes it quite static in nature. Of course, nothing prevents us from registering new

object implementations with the object adapter, but this service is not facilitated in any special way by our MSA.

We consider that the idea presented in the MULTE project, of extending the object adapter of an ORB with streaming capabilities would provide better support for dynamic QoS. The disadvantage would of course be, the need to port the enhanced object adapter to every new release of the system used.

2. **Evolution of QoS requirements.**
   As far as support for the evolution of QoS requirements is concerned, we consider our implementation to be quite flexible. It is for instance easy to add new attributes to the FIDL++ language, and the list of existing ones is already quite comprehensive. It is also possible to give different meanings to the same attribute in the same listing. The specification for our MSA demo application, presented in Program Listing 7.1, is an example of this. Here we use the samplerate attribute with both the image and the audio media element declaration. For the audio element, its meaning really is that of the rate with which the sound is sampled, but for the image declarations we use it as a substitute for *framerate*. We had as a goal to remain as compatible as possible with the original FIDL language, and we were therefore reluctant to extend the language with new keywords. Of course, if new multimedia data appears, and it exhibits characteristics which can not be expressed with only the existant FIDL attributes, then this would be a good reason to introduce new ones. The two attributes we have allowed ourselves to add to the language, consumerElement and producerElement, have not been added because of the evolution of QoS requirement, but as a means to allow our compiler to generate the interface to the MEMs.

   In addition, we note also that FIDL was introduced already in 1997 [16], and in the 7 years which have passed since, we have not seen the appearance of any *new* types of multimedia data, which warrented the extension of FIDL for our work in 2004. See the discussion of multimedia data types in Section 2.1.2.

3. **Transparency versus fine grained control.**
   Some of the explanations we have given for point 1 of this list apply to this point too. Our specification of QoS requirements is very fixed, in the sense that it is limited to what FIDL can handle. However, it is very comprehensive.

   While the developer can not specify that he or she desires "good video quality", as the requirement states, we argue that *the developer* does not really need to address QoS at this "application" level.

As far as the users of the system are concerned, we consider that it is useful for many of them to relate to such general characterization of QoS. This is not a feature directly supported by our MSA implementation, but we argue that it is easy to provide the necessary mapping from such high level QoS requirements to those needed by our FIDL based implementation. Our MSA demo application, is a proof of this point.

For this application we have created a GUI, which maps the user's choices of QoS parameters to, on the one hand calls to our streaming API, and on the other hand, to certain values of the parameters which control the streaming of data. The labels associated with our GUI controls are targeted to developers more then to general users, but it would be easy to change a few strings in our program, to say High video quality instead of QOSL 1(VGA+Mic), Medium video quality instead of QOSL 2(SIF+Mic) and Low video quality instead of QOSL 3(QSIF+Mic). It is easy to make as many calls to our streaming API, based on a singel user action. In our application, the radio buttons associated with the QoS levels do not only change the size of the picture, but also resets all other QoS parameters to their default values. Thus the framerate is set back to 5, and the sound is set to mono, 8000 samples per second, 8 bits per second. This shows how easy it is to provide the user with QoS choices in a language they are familiar with (high quality, medium quality, etc.) and still have our system use only the streaming API developed so far.

4. **Policy control.**
We have not addressed the issue of policy control. In the case of our demo application we continually monitor the bandwith usage, as we make changes to the QoS configuration of the streaming object. The opposite would be just as easy to implement, for certain simple scenarios. For instance, the application can monitor the achieved bandwidth usage and compare it to the theoretical bandwidth needed to stream at a given QoS level, and sublevel[1]. If the available bandwidth is not enough the application can automatically adapt, *in a simple way*, by going down to the highest achievable QoS level. However, in our dealing with levels and sublevels of QoS, we have only provided a simple ordering among them, based on reading the attributes of media element declarations in a top-down fashion (first come, first served) and by reading the possible values of the attributes in a left-to-right fashion: the first value is the most preferred, the last one the least preferred. The possibility to specify more sophisticated policies would be a useful feature.

[1] QoS sublevels are addressed in Section 7.8

5. **Automatic support for compatibility control.**
   This is not an issue for our work. We refer to implementations of such functionality in [16] and [25]. In this thesis we start from where the implementation of the Intersector of [16] has left. Our FIDL++ specifications *are* the result of a compatibility check, and our compiler uses this property of the specifications to generate code which implements a streaming API exhibiting these *compatible* characteristics.

6. **Support for seamless system evolution.**
   This issue is too complex for a work of this thesis' duration. Our solution does not allow the alteration of the stream objects it creates from a FIDL++ specification. It is not clear to this author if it is possible at all to let a C++ program evolve in the manner implied by this requirement. If it is, we have not studied the mechanisms necessary to achieve that behavior. In programs implemented with interpreted languages, it is easier to change the code of the system while the system is operating.

The general requirement posed to our work, was to provide a solution which allows us to treat streams of (multimedia) data as first class objects – in our case C++ objects. This we have achieve, by means of the stream object we create, from a FIDL++ specification. In an MSA based application, the programmer can call methods on this object, which affect the behavior of the whole streaming process. All our API calls are directed to this object, which makes our solution easy to use.

No one has told us what the streaming API should be like. Some choices, like the general operations introduced in Section 10.1 have been obvious, the others we have suggested ourselves. We argue now that the API proposed by us is both simple and intuitive in regard to method names.

A question which we can not really answer yet, is weather this API is enough? Probably not. Future use of the system will reveal new features which would be useful to automate in a MSA.

A very nice feature of our solution is that it is compatible with any version of ICE. By means of a our compiler, we actually provide native Slice compatibility, in spite of the use of FIDL++.

12.3 Summary

In this chapter we have presented an evaluation of our work. We have pointed out what we consider to have achieved, and some of the requirements which we did not met.

We conclude this thesis with the next chapter.

Chapter 13

Conclusion

In this chapter we present a summary of our work, a short discussion of the degree to which we have managed to achieve our goals and point out some suggestions for further work.

13.1 Summary of our work

The general goal of this thesis, stated in Section 1.3, is to provide a way to treat streams and flows of (multimedia) data as first class objects in the context of object-oriented middleware based applications.

In order to present a solution for this research topic, and based on the history of this thesis' project, we have worked on two systems:

1. A Da CaPo core implementation.
2. A MSA, design and implementation.
   (a) fidl, a compiler for FIDL++.
   (b) A streaming API.
   (c) A MSA run-time

In connection with our systems, we have implemented 3 demo applications, presented in Chapter 11:

1. A Da CaPo demo.
2. A general ICE based demo, using the polling client approach.
3. A MSA demo, which is our main demo application.

13.2 Goals - did we reach them?

We summarize now the goals we had defined for our thesis, and what we have done to achieve them. A more thorough discussion, in the light of *requirements*, is given in Chapter 12.

1. **Da CaPo core implementation.**
   We have implemented a Da CaPo core. It is written in C++, and it is independent of any other systems. We argue that our implementation provides a great deal of flexibility and extendability. In addition it also has a very simple module interface. However, this implementation is only a core. Especially, we have not had time to look at the issues of dynamic adaptability of applications base on Da CaPo. We argue that our implementation can be extended to provide this feature too.

2. **Establish Requirements for an middleware based MSA.**
   We have specified the requirements for our work, both those which were imposed on us by the MULTE project, and those which we have imposed ourself.

3. **MSA implementation.**
   We have designed the run-time of a MSA, and have implemented a compiler for a slightly modified version of FIDL. By means of this compiler we are able to parse high level stream and flow descriptions, and to generate from them Slice and C++ code which implements a hierarchy of objects, within regular ICE applications. On the top of this hierarchy, we have an object which encapsulates the behavior described by a FIDL++ description. This object provides a streaming API, by means of a set of methods. In this way a stream of data which can be described by means of FIDL++, appears to the programmer as a regular C++ object.

4. **The integration of Da CaPo and the MSA.**
   We did not have time to deal with this issue. However, we consider that the two systems can be integrated into a common implementation. We mention more details in the section about further work.

13.3 Futher work

Most of the further work we have in mind, after finishing this thesis, has to do with implementing new features into our systems.

13.3.1 Da CaPo enhancements

Adaptability

First of all, we have not implemented an application which proves that our Da CaPo core implementation is suitable for dynamic adaptive behavior of distributed applications. Still, as we have argued in Section 12.1, we consider that the extremely simple module interface of our implementation, combined with the fact that the modules are oblivious of their position in a Da CaPo graph, will allow applications based on our Da CaPo core to exhibit this behavior. Modules can be inserted and removed from protocol graphs by a simple (re)assignment of the variables which point to the buffer(s).

Control management

Another missing feature of our Da CaPo core implementation is the fact that we have not designed and implemented an interface for control management. Our current design is based on the fact that the modules are running in their own threads. A thread can not be started and stopped to pause many times. As such, we have implemented in the modules a certain amount of control management, by making them yield the CPU when they are idle. Thus if the A-module on the producer side takes a break and does not produce packets for a while, the whole protocol graph will bring itself to a state of active waiting. When this module begins to produce packets again, the whole graph will start working again.

One question is by what means should control management be implemented in Da CaPo? We argue that the easiest way would be by means of an existing object-oriented platform for distributed computing, like CORBA or ICE.

13.3.2 MSA enhancements

Batched compilation of FIDL++ descriptions

So far we have not considered the scenario where our compiler is supposed to compile several FIDL++ files residing in the same directory.

When such a need appears, it will be necessary to have the interface implementing code derived from each FIDL++ specification into a separate files. As it is now, the compiler always calls the files where the interface implementation code resides for interfaceImplementations_callback.h and interfaceImplementations_callback.cpp. This policy would lead to the overwriting of these files, with only the last FIDL++ description's files remaining.

More control of attribute assignments

In our implementation of the compiler, we do not make active use of the information contained in the attribute specifications of the media elements of a FIDL description. The use we do make, is to retrieve the "default" value of each attribute, and use it for initializing purposes. By default value of an attribute we mean its left most value. However, an assignment like:

```
image VGA {
  producerElement = "EWebCam";
  consumerElement = "EImageViewer";
  samplerate = (5, 30);
...
};
```

suggests that the samplerate attribute can have values between 5 and 30, with 5 as its default value. As part of our streaming API, we can call the

```
setEWebcamsamplerate(int samplerate);
setEImageViewersamplerate(int samplerate);
```

In our current implementation, there is made no check to what integer values we give as parameters to these operations. This should be dealt with. One way is to store all the values an attribute can have, and establish a policy for dealing with the situations when a program attempts to assign an illegal value. If there is an ordering among the values, we cam always use the least and the greatest values as escape values. If there is no ordering among the values, we can use the default value as an escape value.

Ideally, an illegal assignment should be discovered at compile time. We have entertained the idea of declaring a C++ data type for each possible value of an attribute. For the samplerate attribute mentioned above, we would declare integer constants for all values from 5 to 30. The approach is somehow messy, but since our compiler does all the work behind the scenes, it would be implementable.

We have not implemented this tight attribute assignment control because we have focused on other more essential features of the MSA and because we can assure that our API calls always receive legal values for parameters by carefully designing a GUI. In our MSA demo application, for instance, we enforce the 5 to 30 range of values for the images' samplerate attribute, by enforcing it on the slider and spin box elements used, as shown in Figure 11.4.

Enhanced API: getAttribute(), setAll()

It is common practice for the design of an API to provide get methods alongside with the set methods which are declared, so that the programmer can inquire of the value of a certain variable. Providing get methods as part of the MSA's API is not very difficult, since the hard work of parsing the FIDL specification and traversing the data structures generated during

parsing has already been done in connection with the code generation process presented in this thesis. We have not done it because it is not absolutely necessary in order for us to build a proof of concept demo application.

Another usefull method which could have been added to our streaming API is a setAll method for each flow implementation. This would be useful if we want to set several attributes at once. We could then send all of their values in a single invocation, instead of an invocation per attribute setting, as it is the case now. If we want to move from sound with lowest quality to sound with highest quality in our demo application, we need to send three separate invocation, which each carry a singe integer value. The ratio of payload data to header is significantly high in this scenario.

Run-time error management

A weakness of our implementation of the MSA is that we have not designed special mechanisms for error detection and control. The work with the generation of code has been so time consuming that we have prioritiezed to implement a working system, rather then equip it with even such useful features as error management and control. I this way we could at least prove the feasibility of our approach.

A stream object can be brought into its Undefined state by many possible network failures. Some kind of mechanism should be developed, to ensure that a stream object detects that is has been brought into the Undefined state. The objects can then attempt to rectify the situation. Ideally, the stream object should try to bring itself again into a legal Streaming state. If it does not succeed to do so, it should bring itself into the Stopped state. The reason why the Undefined state is unacceptable is that it gives no accurate description of what is working and what is broken among the stream object's composing elements.

ICE supports the mechanism of throwing and catching errors at the level of Slice declarations. This feature is the primary means we propose to address the issue. We should enhance the Slice code generated by our compiler with appropriate exception types and exception handling operations.

A better object model for the MSA's run time

We have attempted to follow good object-oriented principles in our implementation. Nevertheless this work is a first implementation of our systems, and we can see now that we probably can achieve the same results as now with a cleaner and simpler class hierarchy. We had a desire to make more use of polymorphism in our implementation, but we could not always use the types declared by slice2cpp from Slice files in the way we desired. This is both because we do not understand well enough the details of the Sliceto C++ language mapping and because we are pushing the limits of the RPC

programing paradigm, which is what ICE is offering.

Other advanced features

Other ways to enhance our implementation would be to make active use of some of ICE's services, like Glacier which allows end-points to communicate through firewalls.

Ice also offers native supportuse for other protocols like UDP and SSL. We could investigate how to take advantage of these protocols' properties, so that our MSA could function in several modes and be offering an API adapted to th e used communication protocol.

A third idea would be to investigate how we could implement our MSA on Real-Time Linux, provided that ICE runs on that operating system too.

13.3.3 Integrating Da CaPo and the MSA

Probably the greatest weakness of this thesis is that we have not had time to investigate how Da CaPo and the MSA could be integrated into a common system. However, by working with both of them, we have a certain understanding of how this could be accomplished. The basic idea is to

1. create a common object which supports both Da CaPo's interface and the interfaces required by a MEM object
2. replace the generic network used by Da CaPo with an instance of a MSA.

This is illustrated graphically in Figure 13.1, where we have put together the systems presented in Figures 8.1 and 6.1. Of course the MSA object in Figure 8.1 is only a simplified representation of Figure 9.7.

13.4 Summary

We have seen in this chapter, that we have achieved most of the goals for this thesis. We have also pointed out what we consider is still lacking, and we have presented several suggestions for further work.

Working with this thesis has been both challenging, interesting and very educative. Many thanks to everyone who has contributed at its completion.

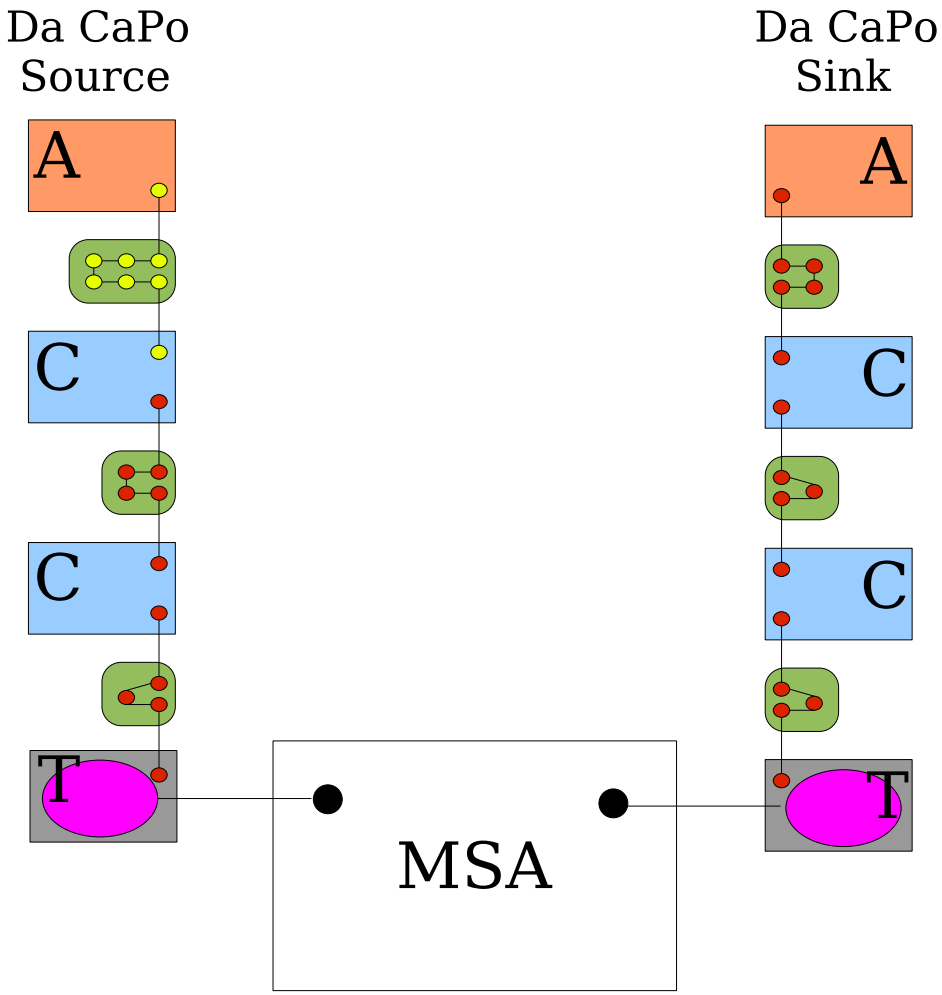

Figure 13.1: *Integrated Da CaPo and MSA system.*

Appendix A

How to build MSA based applications

In this chapter we provide a step-by-step description of how to build an application using the MSA. As a specific example we use our demo application.

1. **Base objects**:
   The classes in the base directory of our source code must be compiled to object files before attempting to compile anything else. We think it is a good thing to do it first. In the base directory we execute

   ```
   $ make objects
   ```

2. **FIDL specification**:
   Provide a FIDL description for your application at an *appropriate location*. We use an empty directory, and copy the webcam.fdl file there. An *appropriate location* must for now be a directory on the same level with the base directory in our source code. See the placement of the demo directory in our source code in relation to the location of the base directory.

3. **Parse your FIDL specification**:
   For our demo we execute

   (Have I changed the name from FIDL to MSA?)

   ```
   $ python2 ../parser/fidl.py all webcam.fdl
   ```

   The PLY tools we have used for the parser are written for python2, so we use python2 instead of python, but on our Fedora Core 2 test machine python works just as well.

   ../parser/fidl.py is the location and the name of our compiler.

   Following the name and location of our compiler is a compulsory argument which tells our compiler which file to generate. If you omit this argument the compiler will exit immediately.

If you remember, we presented the following summary of files generated by our compiler, at the end of Section 8.1:

1. webcam.ice
   Slice definitions to be compiled by slice2cpp.
2. interfaceImplementations_callback.h
   Header file for the C++ implementations of the Slice definitions from webcam.ice.
3. interfaceImplementations_callback.cpp
   Code file for the C++ implementations of the Slice definitions from webcam.ice.
4. serverCB.cpp
   Basic ICE server application. Fully functional, but the application logic can be extended.
5. clientCB.cpp
   Basic ICE client application. The application logic must be added to this file. Optionally, the client application can be hooked to a GUI.
6. config.server.callback
   Configuration file for the server application.
7. config.client.callback
   Configuration file for the client application.
8. Makefile
   Makefile to compile all of the above.

If the argument has the all value, as shown above, all these files will be generated. If you want to (re)generate only specific files, provide their number instead of all. all is the same as 12345678, or any other number containing all the digits from 1 to 8.

This feature of the compiler is probably useful only during the debugging process of the compiler itself. It has been added to the compiler because the compilation time for files 1 to 3 is quite long compared to the other files, and by not regenerating them more often then absolutely necessary, we do not have to compile them more times then absolutely necessary.

Finally, the last argument is the name of the file to be compiled.

4. **Compile the Slice code**:
   If you do not have a GUI ignore this step. If you use a GUI, it might have dependencies on the header file which will be generated from the webcam.ice file by the native ICE compiler, slice2cpp. Run therefore your Slice file through slice2cpp now. If you do not have a GUI which

depends on this header, ignore this step, as it will be executed by the main Makefile when necessary.

We generate this header file now by calling:

```
$ slice2cpp webcam.ice
```

5. **Provide a GUI**:
   If you do not have a GUI ignore this step. If you want a GUI for any of the client or the server applications, you will need to add the GUI files to the location directory, no later than by the time you attempt to compile. We have a GUI for our client application, so we copy the contents of the gui directory to the location directory now.

6. **Compile the GUI**:
   If you do not have a GUI ignore this step. Because there are interdependencies between the code in the interfaceImplementations_callback.h and interfaceImplementations_callback.cpp files and the GUI files, and because a particular GUI is unlikely to fit several applications, the GUI files must first be copied where the interfaceImplementations_callback.h and interfaceImplementations_callback.cpp files are.

   Since the compiler can not know in advance what GUI you will provide for the particular FIDL specification you are compiling now, if any, and because it can not guess what you will name the files which will contain the GUI, the compiler can not generate targets to compile the GUI when it generates the Makefile for the whole project. You must therefore do what ever it takes to compile to object files the GUI you have provided.

   For our demo application we have a separate makefile called gui-Makefile which only compiles our GUI to object files. The objects which implement the functionality of the media elements will link against these object files at link time. We call make with this file as an argument:

   ```
   $ make -f gui-Makefile
   ```

7. **Compile the application**:
   You are now ready to compile the application. Since our compiler has generated a Makefile specifically for this FIDL specification, we call make on it:

   ```
   $ make
   ```

   The server application is compiled to an executable called serverCB and the client application is compiled to an executable called clientCB.

   However, the client and the server applications will do "nothing", except for initializing a whole lot of objects. While the server application

is *complete* in the sense that it is ready to serve a client which behaves according to the pattern described in the webcam.fdl specification, the client application logic is missing, because you can use the infrastructure set up by our compiler in a thousand ways.

You can alter the application logic of both the client and the server applications, but you will probably only want to work on the client application. In both the serverCB.cpp and the clientCB.cpp files the compiler provides a *safe* place where you can write your application logic. This place is found toward the end of the files, between these comment lines:

```
// begin user code=============================

// end user code===============================
```

You are welcome to write your application logic some other places in these files too, if you know what you are doing. At this point, however, all objects are guaranteed to be initialized and this location is also placed before any clean up code which might be necessary to be run before the applications exit, so it is a safe place to work at.

For our demo application we have a very simple client side application logic:

```
// begin user code=============================

  IceUtil::ThreadPtr
    gui = new MainGUI(argc, argv, stream,
                      EImageViewervar,
                          ESoundCardvar);
  threads.push_back(gui->start());

  int res = 0;
  res = stream->setQOSLevel(3);

  // end user code=============================
```

As you can see, most of the application logic is concerned with firing up the GUI. We request then that the stream object will operate at QoS level 3, and we can use the res variable to see if the request has been granted.

The application logic of our demo application is deceivingly simple. We have placed tons of GUI functionality in the GUI files, which you might want to, and which you also can code here, in the clientCB.cpp file, if you prefer.

Normally you would make calls to our streaming API here. As you can see we pass stream, the pointer to the stream object, to the main GUI

object. In this way all user actions will be mapped by the GUI to calls to our streaming API.

See the mainWindow.cpp file for details on how we make calls to the streaming API. As you will see, the calls are very simple, and most of the code is concerned with handling the GUI elements. As examples we present the implementation of the following four Qt slots:

```
void
MainWindow::startStream() {
  streamPtr->stream();
}

void
MainWindow::stopStream() {
  streamPtr->stop();
}

void
MainWindow::pauseStream() {
  streamPtr->pause();
}

void
MainWindow::updateFramerate(int fr) {
  cout << "MainWindow::updateFramerate(). " <<
      fr << endl;

  if (!streamPtr->setEWebCamsamplerate(fr))
    frSpinBox->setValue(frameRateOld);
  else
    frameRateOld = fr;
}
```

8. **Certificates**:
   Before you attempt to run the executables, copy the whole certs directory to the location directory. Notice that you need to copy the whole directory, not only the files from it, as it was the case for the GUI directory. The reason why we need these files is that the code we generate for our executables is half way initialized to use the SSL (Secure Socket Layer) protocol as a communication's protocol, instead of TCP. Without these files, located in a directory called certs where the executables are run from, the executables will not start.

Bibliography

[1] Gordon Blair and Jean-Bernard Stefani. *Open Distributed Processing and Multimedia*. ADDISON-WESLEY, 1998. ISBN 0-201-17794-3.

[2] Peter J. Denning, Douglas E. Commer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. *Computing As A Discipline*, 1989. Condensed article from the *Report of the ACM Task Force on the Core of Computer Science*. 1989 ACM 0001-0782/89/0100-0009.

[3] Tom Fitzpatrick. *Open Component-Oriented Multimedia Middleware for Adaptive Distributed Applications*. PhD thesis, Computing Department Lancaster University, September 1999.

[4] Daniel g. Waddington, Geoff Coulson, and David Hutchison. *Specifying QoS for Multimedia Communications within Distributed Programming Environments*.

[5] Andreas Gotti. *The Da CaPo Comunication System*. Swiss Federal Institute of Technology (ETH Zürich), Computer Engineering and Networks Laboratory (TIK), CH - 8092 Zürich, June 1994. Technical Report.

[6] The Object Management Group. *Audio/Video Stream Specification*. The Object Management Group, January 2000. New edition, no text changes since June 1998.

[7] The Object Management Group. *The Common Object Request Broker: Architecture and Specification*. The Object Management Group, December 2001. Revision 2.6.

[8] Michi Henning. *Massively Multiplayer Middleware*. Published in ACM Queue Magazine (Vol 1, Issue 10, February 2004). Article available from *www.zeroc.com*.

[9] Michi Henning. *A New Approach to Object-Oriented Middleware*. Published by the IEEE Computer Society, January-February 2004, IEEE

Internet Computing. Article available from *www.zeroc.com* and from *www.computer.org/internet/*.

[10] Michi Henning and Mark Spruiell. *Distributed Programming with ICE.* ZeroC, Inc., May 2004. Revision 1.4.0, with contributions from Benoit Foucher, Mark Laukien, Matthew Newhook, Bernard Normier. Downloadable from www.zeroc.com.

[11] Michi Henning, Mark Spruiell, Marc Laukien, and Matthew Newhook. *Distributed Programming with ICE.* ZeroC, Inc., July 2003. Revision 1.1.0, downloadable from www.zeroc.com.

[12] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++.* Addison-Wesley, 1999. ISBN 0-201-37927-9.

[13] ISO/ITU-T. *ITU-T X.901 | ISO/IEC 10746-1 ODP Reference Model Part 1. Overview*, May 1998.

[14] ISO/ITU-T. *ITU-T X.901 | ISO/IEC 10746-1 ODP Reference Model Part 2.*, May 1998.

[15] Tom Kristensen and Thomas Plagemann. *Enabling Flexible QoS Support in the Object Request Broker COOL.* Center for Technology at Kjeller (Center For Technology at Kjeller), University of Oslo http://www.unik.no(tomkri, plageman).

[16] Sindre Medhus. *FIDL.* University of Tromsø, Norway, 1997. Master's Thesis.

[17] Microsoft. *Getting started in .NET.* http://www.microsoft.com/net/.

[18] University of Chicago. *PLY (Python Lex-Yacc).* Available from *http://systems.cs.uchicago.edu/ply/*.

[19] Terrence John Parr. *Language Translation Using PCCTS and C++.* Automata Publishing Company, San Jose, CA 95129, 1993. ISBN 0-9627488-5-4, freely distributed on Internett as well.

[20] Terrence John Parr, H. G. Dietz, and W. E. Cohen. *PCCTSReference Manual, Version 1.00*, August 1991. Latest release is 1.10.

[21] Thomas Plageman[1], Frank Eliassen[2], Brita Hafskjold[3], Tom Kristensen[1], Robert H. Macdonald[3], and Hans Ole Rafaelsen[4]. *Flexible And Extensible QoS-Management For Adaptive Middleware.* [1]University of Oslo, UniK - Center for Technology at Kjeller {plageman, tomkri}@unik.no, [2]University of Oslo, Department of Informatics frank@ifi.uio.no, [3]Norwegian Defence Research Establishement {bha, Robert-H.Macdonald}@ffi.no, [4]University of Tromsø, Department of Computer Science hansr@acm.org, 2000.

[22] Thomas Plagemann, Vera Goebel, Pål Halvorsen, and Otto Anshus. *Operating System Support for Multimedia Systems*, 1999. To be published in: Computer Communications Journal, Special Issue on Interactive Distributed Multimedia Systems and Telecommunications Sservices 1998(IDMS'98), Elsevier Science, Winter 99.

[23] Thomas Peter Plagemann. *A Framework for Dynamic Protocol Configuration.* Swiss Federal Institute of Technology (ETH Zürich), Computer Engineering and Networks Laboratory (TIK), Diss. ETH No 10830, 1994. Dissertation for the degree of Doctor of Technical Sciences.

[24] Hans Ole Rafaelsen and Frank Eliassen. *Trading and Negotiating Stream Bindings.* Published at IFIP/ACM Middleware2000, New York, April 2000. hansr@cs.uit.no, frank@ifi.uio.no.

[25] Hans Ole Rafaelsen and Frank Eliassen. *Trading Media Objects with CORBA Trader.* hansr@cs.uit.no, frank@simula.no.

[26] Douglas C. Schmidt and Steve Vinoski. *Object Adapters: Concepts and Terminology.* Object Interconnections, Column 11, published in the October 1997 of the SIGS C++ Report magazine.

[27] Ralf Steinmetz and Klara Nahrstedt. *Multimedia: Computing, Communications & Applications.* Prentice-Hall, 1995. ISBN 0-13-324435-0.

[28] Chorus Systems. *CHORUS/COOL-ORB Programmers's Guide.* CS/TR-96-2.1.

[29] Chorus Systems. *CHORUS/COOL-ORB Programmers's Manual Reference.* CS/TR-96-3.2.

[30] W3C. See http://www.w3.org/2000/xp/Group/ and http://www.w3.org/TR/.

[31] James Won-Ki, Jong-Seo Kim, and Jae-Kyu Park. *A CORBA-Based Quality of Service Management Framework for Distributed Multimedia Services and Applications.* Published in IEEE Network, March/April 1999.