An open architecture to support distributed services in a digital television studio environment

Pedro Miguel Vieira Alves Ferreira

Dissertação submetida para satisfação parcial dos requisitos do grau de mestre

em

Engenharia Electrotécnica e de Computadores

(Área de especialização de Telecomunicações)

Dissertação realizada sob a supervisão do Professor Doutor José António Ruela Simões Fernandes,

do Departamento de Engenharia Electrotécnica e de Computadores
da Faculdade de Engenharia da Universidade do Porto

Porto, Setembro de 2000
An open architecture to support distributed services in a digital television studio environment

Pedro Miguel Vieira Alves Ferreira
Abstract

The evolution of television during the recent years is presenting new requirements in terms of the integration of the components of a modern television studio.

A variety of new solutions and equipment, mostly derived from the information technology industry, is being deployed in the television studios. However, integration is usually made in an *ad-hoc* basis, almost always retrofitted in order to be compatible with the earlier technology and legacy systems.

This dissertation addresses this problem by presenting a new architecture for dealing with the management, command and control of devices and tools called Distributed Middleware for Multimedia Command and Control, which is completely based on distributed object-oriented computing techniques, namely the Common Object Request Broker Architecture (CORBA).

Areas focussed on by this solution are the global management of all studio components, the command and control of such entities, the management of network connections, the reporting of events and alarms and the linking between essence and metadata.

Having had its inception during the European Commission funded ACTS ATLANTIC project, this architecture is being used and developed within the ORBIT project, a joint effort of the British Broadcasting Corporation Research & Development Department (BBC R&D) and the Instituto de Engenharia de Sistemas e Computadores do Porto (INESC Porto).
A evolução da televisão nos últimos anos tem criado novos requisitos com vista à integração dos diversos itens que compõem um estúdio moderno.

Uma grande variedade de novas soluções e equipamento, na sua maioria derivada da área das tecnologias de informação, está a ser instalada nos estúdios de televisão. Contudo, a sua integração é geralmente efectuada de forma algo precária, quase sempre limitando as suas possibilidades, no sentido de manter a compatibilidade com a tecnologia legada pelas anteriores instalações.

Esta dissertação discute este problema propondo uma nova abordagem à gestão e ao controlo de dispositivos, abordagem esta completamente baseada em técnicas de sistemas distribuídos orientados ao objecto, nomeadamente na Common Object Request Broker Architecture (CORBA).

As áreas abrangidas por esta solução são a gestão global dos componentes e dispositivos presentes num estúdio, bem como o seu controlo; a gestão das ligações feitas através da rede; a notificação de eventos e de alarmes; e a associação entre essência e metadata.

Tendo tido a sua concepção inicial no projecto ACTS ATLANTIC, esta arquitectura está actualmente a ser explorada e desenvolvida no projecto ORBIT, um esforço conjunto do departamento de investigação e desenvolvimento da British Broadcasting Corporation (BBC R&D) e do Instituto de Engenharia e de Computadores do Porto (INESC Porto).
L'évolution de la télévision a crée, dès les dernières années, de nouvelles nécessités en ce qui concerne l’intégration des composants d’un studio de télévision moderne.

Une grande variété de nouvelles solutions et d’équipage, dérivée principalement du domaine des technologies de l’information, est en train d’être installée dans les studios de télévision. Cependant, l’intégration est généralement faite d’une façon précaire, vue que, dans la plupart des cas, ses potentialités sont limitées afin de maintenir la compatibilité avec la technologie antérieure.

Cette dissertation aborde ce problème en proposant une nouvelle vision face à la gestion et au contrôle de dispositifs, cette-ci basée fondamentalement sur des techniques de systèmes distribués orientés à l’objet, plus précisément sur la Common Object Request Broker Architecture (CORBA).

Les domaines compris par cette solution sont la gestion globale des composants et des dispositifs présents dans un studio, et aussi leur contrôle; la gestion des connections faites à travers le réseau; la notification d’événements et d’alarmes; et l’association entre l’information audiovisuelle et la métainformation.

Ayant eu son commencement avec le projet ACTS ATLANTIC, cette architecture est actuellement en train d’être appliquée et développée dans le projet ORBIT, un effort conjoint de la British Broadcasting Corporation (BBC R&D) et de l’Instituto de Engenharia e de Computadores do Porto (INESC Porto).
During the recent years, the evolution of television technology and the new services made available placed a new set of requirements to the management and operation of television studios.

The new requirements on minimizing the time spent in production, the need to produce faster and for more channels, the re-purposing of content for different media like standard definition television, high definition television, cinema or the Internet, the increased competition and the urge in reducing costs, lead to the need to maximize the power of the available technology.

The technology itself is evolving with existing products being enhanced, along with a growing use of solutions borrowed from the information technology world, like the appearance of advanced disk-based media servers, more sophisticated non-linear editors, advanced asset management tools and the ever-increasing use of compression techniques like MPEG-2.

However, an enormous lack of integration between products of different manufacturers and between different operational areas within a company is still apparent. But to reconcile these differences is a daunting task in an environment where investment in equipment and tools is measured in millions.

Several companies and institutions are currently working on solving these problems, either by developing new products or by defining new standards and recommendations.
This dissertation contributes to this effort by proposing an architecture for the seamless command and control of the different components that exist on a digital television studio.

This work is completely based on the use of distributed object-oriented techniques, one of the areas of major evolution in the arena of information technology, and was the result of the application of these techniques in two projects directly related to television applications.

**Organization of the Text**

The first chapter introduces the reader to the evolution of television and is followed in the second chapter by an overview of the relevant work done on standards bodies.

The third chapter gives an overview of the projects that led to this work, along with a similar and pertinent effort.

The fourth and fifth chapter discuss in detail the analysis, design and implementation processes of the aforementioned architecture.

Finally, the sixth chapter goes over the main points and tries to foresee the evolution of distributed object-oriented based control applied to the television.

**Acknowledgements**

An undertaking such as this one is not the result of a single person’s work, but rather a collective effort of several people and institutions.

I am especially grateful to INESC Porto for giving me the opportunity to collaborate with projects as interesting as the ones I have been involved with, and in particular to Prof. Artur Pimenta Alves for his commitment to the quest for new projects in this area.

I obviously owe a lot of gratitude to Prof. José Ruela, who, besides leading my academic career for the last few years, invited me to join INESC Porto’s Telecommunications Unit, giving me the chance to be immersed in such an overwhelming environment.

A great word of thanks also goes to everyone who was involved with the ATLANTIC, ORBIT and VIDION projects. Besides the excellent learning experience, it has been a real pleasure to work in these projects.
Of course, a great thanks to all members of the brand new INESC Porto’s Media Objects Group, where technical competence is only surpassed by human skills. A special mention goes to Pedro Cardoso, for always making sure that we have the necessary resources for doing our job properly and for his constructive comments and suggestions to my work; and to Vítor Teixeira, Ernesto Santos and Ricardo Morla, for their help in designing and implementing DIMICC and DETAIL, as well as for their feedback during our “feet-on-the-desk” brainstorming sessions. Sorry about the bugs!

A word of gratitude is also due to the BBC for funding ORBIT and for believing in our competence, as well as in the solutions we advocate. The contributions of Phil Tudor and Peter Brightwell were especially important for this work, due to their insightful comments and new ideas.

Finally, a huge thank you goes to my family, especially to my parents, for giving me everything I ever needed, and to Susana, for being always there, even when I am away or too much busy to give her the attention she deserves.

**Reader Notes**

The font used in normal text is Times New Roman. **Courier New** is used for computer programmes or class names.

Several acronyms are used across the text. In order not to clutter the text with long definitions, the reader is asked to refer to the acronym list on the beginning of this text.

Bibliographic or Web references are listed at the end of the document. Whenever referred to on the text, a number between parentheses will identify them.
Table of Contents

ABSTRACT ................................................................................................................... III
RESUMO ...................................................................................................................... IV
PRÉCIS ........................................................................................................................ V
PREFACE ..................................................................................................................... VI
Organization of the Text ............................................................................................ vii
Acknowledgements ...................................................................................................... vii
Reader Notes ............................................................................................................... viii
TABLE OF CONTENTS ............................................................................................. IX
TABLE OF FIGURES ................................................................................................. XIII
ACRONYMS ................................................................................................................. XV

CHAPTER 1 - TELEVISION TECHNOLOGY AND PRACTICES ................................. 1
1.1 - Television Technology Evolution ...................................................................... 2
  1.1.1 - Going Digital ............................................................................................... 2
  1.1.2 - Inertia .......................................................................................................... 3
  1.1.3 - Metadata ..................................................................................................... 4
1.2 - State of the Art on TV Studios ............................................................................ 5
  1.2.1 - Play-out ...................................................................................................... 5
# Table of Contents

1.2.2 - Production ................................................................. 6
  1.2.2.1 - Production Process Overview .................................. 6
  1.2.2.2 - Archive Search ................................................ 7
  1.2.2.3 - Editing ............................................................... 8
  1.2.3 - Metadata Management ............................................. 9

## CHAPTER 2 - WINDS OF CHANGE ........................................ 10

2.1 - The EBU/SMTE Task Force Report .................................... 11
  2.1.1 - Formats .................................................................... 11
  2.1.2 - Metadata ................................................................. 11
  2.1.3 - Wrappers ................................................................. 11
  2.1.4 - Object-Based Control .............................................. 12

2.2 - Distributed Object-Oriented Computing ............................. 13
  2.2.1 - DCOM vs. Java vs. CORBA ....................................... 14
  2.2.2 - CORBA Overview ................................................... 15
    2.2.2.1 - The IDL ........................................................... 15
    2.2.2.2 - The ORB ........................................................ 16
    2.2.2.3 - GIOP .............................................................. 17
    2.2.2.4 - Common Object Services .................................... 18
      2.2.2.4.1 - Naming Service ......................................... 18
      2.2.2.4.2 - Trading Service ......................................... 18
      2.2.2.4.3 - Event and Notification Services ...................... 18
      2.2.2.4.4 - Property Service ....................................... 19
      2.2.2.4.5 - Security Service ........................................ 19
      2.2.2.4.6 - Transaction Service .................................... 20
      2.2.2.4.7 - Audio/Video Streams Service ......................... 20
  2.2.2.5 - Recent Additions ................................................. 20
  2.2.2.6 - Available implementations .................................... 21

2.3 - Standardisation Work .................................................. 22
  2.3.1 - SMPTE .................................................................... 22
    2.3.1.1 - Universal Material Identifier ............................... 22
    2.3.1.2 - Advanced System Control Architecture .................. 23
  2.3.2 - Pro-MPEG Forum ..................................................... 26
  2.3.3 - DSM-CC ................................................................. 27
    2.3.3.1 - Overview ........................................................ 27

## CHAPTER 3 - BACKGROUND PROJECTS .................................. 30

3.1 - ATLANTIC ................................................................. 31
  3.1.1 - Introduction .......................................................... 31
  3.1.2 - Reference Model .................................................... 31
  3.1.3 - Application Control ............................................... 32

3.2 - ORBIT .............................................................. 34
  3.2.1 - Introduction .......................................................... 34
  3.2.2 - Reference Model .................................................... 34
3.3 - DS-CC ................................................................. 36
   3.3.1 - Introduction .............................................. 36
   3.3.2 - Studio Control Overview ............................... 36
   3.3.3 - System Model ............................................. 37
       3.3.3.1 - Device and Tool Registration ....................... 37
       3.3.3.2 - Resource Reservation and Allocation ................ 37
   3.3.4 - Streaming Model ........................................ 38
   3.3.5 - Factory Objects ......................................... 38
   3.3.6 - Events ................................................... 39
   3.3.7 - Studio Components ...................................... 39
       3.3.7.1 - Content Servers ..................................... 39
       3.3.7.2 - Other Components .................................... 40
   3.3.8 - Media and Metadata Management ....................... 40
   3.3.9 - Conclusion ............................................... 40

CHAPTER 4 - DIMICC ARCHITECTURE ................................. 43
4.1 - Context .......................................................... 44
4.2 - Requirements ................................................... 45
4.3 - Analysis and Design .......................................... 47
   4.3.1 - Planes .................................................. 47
   4.3.2 - System plane ............................................ 47
   4.3.3 - Control plane ........................................... 49
       4.3.3.1.1 - Basic Classes .................................... 49
       4.3.3.1.2 - Class Specialisations ............................ 52
       4.3.3.1.3 - Events and Properties ............................ 52
       4.3.3.1.4 - Non-networked Devices Support .................. 54
   4.3.4 - Essence transfer plane .................................. 55
       4.3.4.1 - Connection Management Protocol .................... 55
       4.3.4.2 - Quality of Service .................................. 57
       4.3.4.3 - Multicast .......................................... 57
   4.3.5 - Metadata plane .......................................... 58

CHAPTER 5 - DIMICC IMPLEMENTATION ............................... 60
5.1 - Tools Used ...................................................... 61
   5.1.1 - ACE ................................................... 61
   5.1.2 - TAO ................................................... 62
   5.1.3 - ORBacus ............................................. 63
5.2 - DETAIL .......................................................... 65
   5.2.1 - Overview ................................................ 65
   5.2.2 - Requirements ............................................ 65
   5.2.3 - Analysis and Design ..................................... 66
   5.2.4 - Performance Optimisation ............................... 68
       5.2.4.1 - Memory Copying ..................................... 68
       5.2.4.2 - Concurrency ......................................... 69
   5.2.5 - Testing .................................................. 70
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3 - Component and Service Specialisations</td>
<td>71</td>
</tr>
<tr>
<td>CHAPTER 6 - CONCLUSION</td>
<td>73</td>
</tr>
<tr>
<td>6.1 - Micro-level view</td>
<td>74</td>
</tr>
<tr>
<td>6.2 - Macro-level view</td>
<td>75</td>
</tr>
<tr>
<td>6.3 - The Future</td>
<td>76</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>77</td>
</tr>
<tr>
<td>Books</td>
<td>77</td>
</tr>
<tr>
<td>Papers</td>
<td>77</td>
</tr>
<tr>
<td>Standards</td>
<td>78</td>
</tr>
<tr>
<td>Web Pages</td>
<td>79</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>80</td>
</tr>
<tr>
<td>INDEX</td>
<td>81</td>
</tr>
<tr>
<td>APPENDIX A - OBJECT MODEL NOTATION</td>
<td>83</td>
</tr>
<tr>
<td>APPENDIX B - DIMICC IDL INTERFACES</td>
<td>85</td>
</tr>
<tr>
<td>Types.idl</td>
<td>85</td>
</tr>
<tr>
<td>Encoder.idl</td>
<td>92</td>
</tr>
<tr>
<td>FileServer.idl</td>
<td>94</td>
</tr>
<tr>
<td>MediaServer.idl</td>
<td>95</td>
</tr>
<tr>
<td>StreamServer.idl</td>
<td>96</td>
</tr>
<tr>
<td>VTR.idl</td>
<td>101</td>
</tr>
</tbody>
</table>
## Table of Figures

Figure 1-1 - Production workflow on a typical TV studio ............................................. 6  
Figure 2-1 - A request passing from a client to an object implementation .................... 16  
Figure 2-2 - CORBA local and remote method invocation ......................................... 17  
Figure 2-3 - UMID Format ....................................................................................... 22  
Figure 2-4 - SMPTE ASCA Component Diagram (source: [24]) .................................. 24  
Figure 2-5 - SMPTE ASCA Plane Usage Abstraction (source: [24]) ......................... 25  
Figure 2-6 - SMPTE ASCA Functional Planes View (source: [24]) ......................... 25  
Figure 2-7 - DSM-CC functional model ..................................................................... 29  
Figure 3-1 - The ATLANTIC post-production facility reference model (source: [11]) .... 32  
Figure 3-2 - The ORBIT reference model (source: [13]) .......................................... 35  
Figure 4-1 - System Plane protocol stack .................................................................. 48  
Figure 4-2 - Control Plane protocol stack ................................................................ 49  
Figure 4-3 - Component and VirtualComponent ..................................................... 50  
Figure 4-4 - VirtualComponent, Source, Sink and Endpoint class diagram .......... 51  
Figure 4-5 - Device, NetworkPort and Component ................................................. 52  
Figure 4-6 - Class specialisation example: the VTR package ................................... 53  
Figure 4-7 - The StudioObject interface .................................................................. 54  
Figure 4-8 - A proxy Component for a legacy device .............................................. 54  
Figure 4-9 - Essence Transfer Plane protocol stack ................................................ 55  
Figure 4-10 - Connection establishment sequence diagram ..................................... 56
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-11</td>
<td>Connection teardown sequence diagram</td>
<td>57</td>
</tr>
<tr>
<td>4-12</td>
<td>Metadata Plane protocol stack</td>
<td>58</td>
</tr>
<tr>
<td>4-13</td>
<td>The MediaObject interface</td>
<td>59</td>
</tr>
<tr>
<td>5-1</td>
<td>ACE class hierarchy (source: [41])</td>
<td>61</td>
</tr>
<tr>
<td>5-2</td>
<td>TAO block diagram (source: [42])</td>
<td>63</td>
</tr>
<tr>
<td>5-3</td>
<td>EndpointFactory class diagram</td>
<td>66</td>
</tr>
<tr>
<td>5-4</td>
<td>DETAIL component diagram</td>
<td>67</td>
</tr>
<tr>
<td>5-5</td>
<td>The use of ACE Reactor</td>
<td>69</td>
</tr>
<tr>
<td>5-6</td>
<td>Component dependency graph for hierarchical testing</td>
<td>70</td>
</tr>
<tr>
<td>5-7</td>
<td>Stream Server class diagram</td>
<td>71</td>
</tr>
<tr>
<td>A-1</td>
<td>UML diagram</td>
<td>83</td>
</tr>
<tr>
<td>Acronym</td>
<td>Full Form</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>AAF</td>
<td>Advanced Authoring Format</td>
<td></td>
</tr>
<tr>
<td>AAL</td>
<td>ATM Adaptation Layer</td>
<td></td>
</tr>
<tr>
<td>ACE</td>
<td>Adaptive Communication Environment</td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td>Audio Engineering Society</td>
<td></td>
</tr>
<tr>
<td>API</td>
<td>Application Programmer Interface</td>
<td></td>
</tr>
<tr>
<td>ASCA</td>
<td>Advanced System Control Architecture</td>
<td></td>
</tr>
<tr>
<td>ATM</td>
<td>Asynchronous Transfer Mode</td>
<td></td>
</tr>
<tr>
<td>ATSC</td>
<td>Advanced Television Systems Committee</td>
<td></td>
</tr>
<tr>
<td>BOA</td>
<td>Basic Object Adapter</td>
<td></td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
<td></td>
</tr>
<tr>
<td>DAVIC</td>
<td>Digital Audio Visual Council</td>
<td></td>
</tr>
<tr>
<td>DCOM</td>
<td>Distributed COM</td>
<td></td>
</tr>
<tr>
<td>DETAIL</td>
<td>DIMICC Essence Transfer Already Implemented Library</td>
<td></td>
</tr>
<tr>
<td>DIMICC</td>
<td>Distributed Middleware for Multimedia Command and Control</td>
<td></td>
</tr>
<tr>
<td>DSM-CC</td>
<td>Digital Storage Media Command and Control</td>
<td></td>
</tr>
<tr>
<td>DVB</td>
<td>Digital Video Broadcast</td>
<td></td>
</tr>
<tr>
<td>EBU</td>
<td>European Broadcasting Union</td>
<td></td>
</tr>
<tr>
<td>EDL</td>
<td>Edit Decision List</td>
<td></td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
<td></td>
</tr>
<tr>
<td>GIOP</td>
<td>Generic Inter-ORB Protocol</td>
<td></td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
<td></td>
</tr>
<tr>
<td>Acronyms</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>HDTV</td>
<td>High Definition Television</td>
<td></td>
</tr>
<tr>
<td>IDL</td>
<td>CORBA Interface Definition Language</td>
<td></td>
</tr>
<tr>
<td>IIOP</td>
<td>Internet Inter-ORB Protocol</td>
<td></td>
</tr>
<tr>
<td>IOR</td>
<td>(CORBA) Interoperable Object Reference</td>
<td></td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
<td></td>
</tr>
<tr>
<td>MPEG</td>
<td>Motion Pictures Expert Group</td>
<td></td>
</tr>
<tr>
<td>MXF</td>
<td>Media Exchange Format</td>
<td></td>
</tr>
<tr>
<td>NFS</td>
<td>Network File System</td>
<td></td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
<td></td>
</tr>
<tr>
<td>PES</td>
<td>Packetized Elementary Stream</td>
<td></td>
</tr>
<tr>
<td>POA</td>
<td>Portable Object Adapter</td>
<td></td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
<td></td>
</tr>
<tr>
<td>RMI</td>
<td>(Java) Remote Method Invocation</td>
<td></td>
</tr>
<tr>
<td>SAP</td>
<td>Service Access Point</td>
<td></td>
</tr>
<tr>
<td>SDI</td>
<td>Serial Digital Interface</td>
<td></td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
<td></td>
</tr>
<tr>
<td>SDTV</td>
<td>Standard Definition Television</td>
<td></td>
</tr>
<tr>
<td>SNC</td>
<td>System Notification Channel</td>
<td></td>
</tr>
<tr>
<td>SNS</td>
<td>System Naming Service</td>
<td></td>
</tr>
<tr>
<td>STS</td>
<td>System Trading Service</td>
<td></td>
</tr>
<tr>
<td>TAO</td>
<td>The ACE ORB</td>
<td></td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1 - Television Technology and Practices

The television world has been evolving steadily since the beginning of broadcasting. However, recent years have seen a surge on the pace of development, mostly caused by the increasing synergies between the computing and broadcasting technologies.

These profound changes inevitably affect how people work and deal with the technology. Changes to the usual workflow may or may not be immediate but will eventually be visible.

This chapter discusses this evolution and the impact it is causing on the broadcasting industry.

Bearing this in mind, a critical analysis is also made to the technology and practices normally found on current television facilities.
1.1 - Television Technology Evolution

1.1.1 - Going Digital

The television is known as the box that changed the world. However, the world is also changing television.

From the initial slow pace of evolution, that led the television from the early days of poor quality and availability to the analogue satellites that made thousands of kilometres of separation look like just around the corner, the TV industry and consumers are now facing a rate of development never seen before.

The digital transmission to the home, long envisioned as a major breakthrough but impossible due to the lack of technology, is now almost ubiquitous. Recent standards like the DVB, on its terrestrial, cable and satellite variants, and ATSC are being deployed, or at least will be in the near future. Even if we face once again the cantonment of the Europeans, on the former, and of the Americans, on the latter sides, both use the MPEG-2 compression standard [18] to reduce the bit rates needed to bring high quality television to our homes. Moreover, not only excellent image and sound quality is supported but also interactivity and the sending of ancillary data to the user.

A good example of the new services made available by these technologies is the digital broadcast of Formula 1 races. During these events, the viewer is given access to such information as the current standings and lap times, and is also allowed to select the desired view from a collection of cameras all around the circuit, on the pits and inside the cars.

The acquisition of content, which took a great leap with the use of videotapes that made possible the light portable cameras, is also under great evolution, and from DVs and DigiBetas we are moving towards disk based capture.

But not only the direct broadcast is moving digital. The distribution of movies to the consumers, monopoly of VHS during the last two decades, is itself transitioning to digital. DVD, of its name, is the bit carrier of the near future and, of course, uses MPEG-2 to shrink its size so much as to fit in our pocket.

Even Hollywood, whose need to maintain the reputation of treasurer of the highest quality never allowed the sacrilegious use of rough edged pixels to immortalize divas and gallants, is also being converted by the prophets of the digital. Not only is now ever more difficult
to distinguish reality from synthesis, but work is also underway to replace film with silicon.

Need more examples? Virtual studios with ever changing scenarios; shot changes with fixed cameras; video on demand; pay per view; you name it.

And there is more. The so-called virtual world of the Internet and the Web browser promised to take over the place of television on the role of world changer, and it is. These days, any TV station deserving its name must have at least part of its transmission available on the Net. And here, interactivity is a must.

And if the world is changing, there is plenty of news. And news is only news if one is the first to tell it. Not only dedicated channels like the CNN (with its several variants) or Sky News, but almost every other general TV station, are doing their best to produce the best news as fast as possible and as accurately as possible. This requires highly flexible production and capturing facilities and fast access to archives.

Mergers between companies involved with the press, radio, television and the Web also play an important role here: the synergies between the different facets of the company must increase, and the sharing of content is at premium.

1.1.2 - Inertia

The advances on hardware and on compression techniques, mostly pushed by the IT industry and its massive evolution in power and availability, are changing the way television is seen from the outside. But the inside is still very much the same.

The ability of using new techniques to acquire and produce material is not a panacea to the need to distribute material in so many forms and as fast as possible. The need to reduce costs in order to be competitive in a market like this requires a complete change in the minds of most people who manage and work in TV broadcasters.

Broadcasters are usually huge companies, with millions invested in equipment and workflows incrusted in people's minds. Besides, although the market of the television stations is huge, they represent a small number of costumers for the television equipment manufacturers: only around 4000 television stations and 2000 medium to high quality producers [47]. A small market plus the high quality standards required inevitably equals to very expensive equipment.
This means that the inertia to adopt new technology is very high. Only this can explain the fact that more than 80% of the television stations are still analogue-based.

On top of this, manufacturers, even if they want to sell as much as possible, do not want their customers to think their products have become obsolete long before their cost was amortized, so delaying the deployment of newer technologies is often a choice. Lobbying and the use of proprietary interfaces prevent their places from being taken over by competitors.

1.1.3 - Metadata

There is another word, besides *digital*, that is being ever more used among the television people: metadata. Metadata, or *data about the data*, is ancillary information of the essential data, i.e., the video, audio and data channels. Metadata can be used to describe technical information, such as the bit rate of some video stream or the make of the camera used to shoot a scene; production information, such as the actors involved in a series or the director of a movie; rights information, such as how many times a clip can be used; or pretty much any kind of information that is important to be associated with the data.

The proper and effective use of metadata imposes new requirements to the systems. Of most importance is the adequate integration of the metadata and the audio-visual information to which they are related. Additionally, the need to re-enter metadata should be minimized, by passing it between the different departments, from acquisition through play-out.
1.2 - State of the Art on TV Studios

1.2.1 - Play-out

On a typical TV plant the most important devices are interconnected via a Serial Digital Interface [19] network of point-to-point connections.

SDI is a SMPTE standard that defines a means of connecting base-band digital devices using point-to-point, synchronous connections. Being synchronous, SDI requires all devices to be synchronized to a master clock – the studio clock – in order to work properly.

Routers and switches allow the reconfiguration of the connections, making it possible, for example, to switch the input of the transmitter from a VTR to a studio mixer, when going from a pre-recorded programme to a live show.

Additionally, AES-3 [29] connections are typically used to convey audio signals.

The usual practice is to play out programmes from digital or analogue VTRs and to use disk-based media servers to store commercials and interstitials.

When the programme is to be broadcast in digital compressed form, an encoder is typically included on the path just before the transmitter. That is, even if the programme is going to be compressed, it is usually managed in its base band form.

If metadata is to be inserted on the transmission, or if it will be distributed via a content enhanced medium such as the Web, metadata will typically be semi-automatically fetched from a database or manually inserted by an operator.

Of course, the devices need to be controlled remotely: nobody will be pressing buttons on control panels of devices scattered around a building. The current practice is to use RS-422 connections between a control desk and the devices in order to fulfil the task. Almost every device has such an interface but beware, “the nice thing about protocols is that there are so many you can choose from” [1].

Summing up, we have four different networks - SDI, AES-3, RS-422, and an IT network, such as Ethernet, for metadata – and a set of protocols and databases to live with. Consequently, wiring is complicated and maintenance is, more often than not, a true nightmare.
1.2.2 - Production

1.2.2.1 - Production Process Overview

The diagram on Figure 1-1 shows the typical workflow during the production of a TV programme.

Figure 1-1 - Production workflow on a typical TV studio

After it has been commissioned, the producer outlines the programme and finds out which parts will compose it. During this phase, and in order to reduce the costs involved in shooting new material, the researchers try to find material that can be reused in the archives. In the case of historical documentaries, or whenever the material cannot be shot again, this need is blatant.

After that, there is a phase during which material is rough edited to generate an initial edit decision list (EDL). To keep costs as low as possible, VHS recorders and tapes are used in this phase.

This EDL is then passed to an on-line editing system, where a specialised editor will perform the final editing job.

After the producer approves the programme, copies are sent to the presentation and archive areas and promotions and trailers are generated.

However, each one of these phases has its own problems, many of which are related to the technology used and are dissected below.
1.2.2.2 - Archive Search

In order to ease the searches, archivists typically maintain databases with metadata that helps to find the desired material. Criteria used in the search include keywords, often selected with the help of a thesaurus, that will match the ones introduced by the archivist during the previous annotation of the archived material.

However, it often occurs that the material could not be found using the database, even if it is known to be available. Causes for this range from a poor annotation to the ineptitude of the person who performed the search. In this case, the help of an archivist may be required in order to find the desired items, implying loss of time and resources.

After a successful, or at least satisfactory, search on the database, what the producer gets are pointers to the material, usually in the form of tape identifiers, which will take part on a request to the archive department.

The librarian then searches the archive, usually huge and with every imaginable format of tape and film, and fetches the required items.

For the initial phase of rough editing, VHS copies are generated from the original material but, for the post-production phase, the originals are required. The items are then transported and delivered to the requesters via the internal sneaker-net, that is, the “network” that moves tapes within the station.

The time needed for the production team to get the requested clips is then usually measured in days, escalating the project’s time frame and increasing the probability of tapes being lent directly between colleagues, completely out of control of the archive staff.

Several problems arise here, the first on being the delay and the costs in material and in personnel inevitably introduced by these tasks, plus the more usual than desired loss of tapes or their not being returned in useful time.

Another problem is the myriad of formats usually found in a television archive, mostly analogue ones. Equipment must be available to convert between them and, if their number is not enough, requests may have to be queued until the equipment is free.

All this adds up to sometimes being cheaper to go out and shoot the material again!
1.2.2.3 - Editing

The editing process is often subdivided into an initial phase of rough-cut editing, performed by one element of a production team, and the final craft editing, where the service of a qualified editor is required. Besides, audio editing is almost always performed separately.

In order to be productive, most of the production team own the necessary equipment for rough-cut editing. Since there is a trade between quality and availability, usually cheap VHS VTRs are used. Subsequently, the generated EDL is transferred to the equipment that will be used for the final editing. Depending on various factors, this can be done via the internal network, using a diskette or, lacking a better means, by a printout of the EDL.

The final editing can either be linear or non-linear. Linear editors use analogue or digital VTRs as inputs, perform the switching between inputs and add effects as required.

Non-linear editors are usually computer-based equipment that, after preloading the clips to be edited, allow the user to define switching points and transitions as required. The source media are then used to render the final programme.

Most non-linear editors use proprietary formats to store the video information. The most used formats are variants of MPEG and, mainly, M-JPEG. Of course, being compressed formats, these introduce artefacts that sometimes can be noticed in the final programme. For this reason, the bit rates used are relatively high, in the order of 50 Mbit/s.

If the source material was already in compressed form, a transcoding process must be performed, which will normally degrade the quality of the material.

Things can get worse if later in the chain the programme will be recoded, because artefacts will add up. This happens quite often, for instance, when the broadcast will be made in a compressed format.

When the process is finished, and the programme approved by the producer, it will be transferred to the presentation area. This is usually performed, once again, via the sneaker-net, on a cassette.

One can obviously see some problems in this process. Most striking will probably be the need for several transcoding processes, and the consequent degradation of the quality of the image, but we cannot forget the constant transfer of material and the loss of time and effort on the other tasks.
1.2.3 - Metadata Management

The ever-increasing role of metadata on television was already mentioned. Ideally, metadata should accompany the essence throughout its life within the studio and, at least a selected part of it, should also be broadcast.

However, reality is a little cruder and things just do not work as well as they should.

Metadata is initially entered in one of two places: during the production process, where producers insert metadata relative to technical aspects of the shootings, information about the cast or staff involved in a production, etc.; alternatively, when material is being archived, there is people responsible for annotating all the clips that are going to be stored.

Nonetheless, essence and metadata are usually stored on different places and on different, disconnected, containers. Metadata is commonly stored on databases and sometimes the way it is transferred along the essence is by gluing it (literally!) to the tapes when they are moved.

Of course, the workflow suffers from the lack of coupling between essence and metadata, which sometimes gets out of date or is difficult to find and manage.

During phases like editing, metadata such as rights information are essential for the staff to perform their work. On the other end of the chain, on play-out, sometimes metadata must be reintroduced when it is going to be broadcast along with the essence.
Chapter 2 - Winds Of Change

The first chapter showed how the plethora of new technologies available in the market, associated with the new expectations on the user’s side, are posing new challenges to the broadcasters.

The industry and the research community are working towards the integration of the new technologies and the creation of tools to ease the work of every one involved in this area.

This chapter presents some existing or proposed standards, relevant to this dissertation, which will serve as a basis for the subsequent discussion.

The first section introduces the EBU/SMPTE Task Force Report and discusses its impact in the future of the industry.

The second section presents distributed object-oriented systems as the way to go in approaching the control systems of tomorrow.

Finally, the third section discusses relevant work, currently underway on standards bodies or similar organizations.
2.1 - The EBU/SMTE Task Force Report

On September 1998, the EBU/SMpte Task Force for Harmonized Standards for the Exchange of Program Material as Bitstreams published their final report [46]. This report was anxiously expected and is now seen as the Vision document for the whole industry.

The goal of this task force was to define a set of guidelines for the industry to follow during the coming years, in order to ease their work and to facilitate convergence and definition of new standards.

The aspects focused on the report more relevant to this work are Formats, Metadata, Wrappers and Object-Based Control.

2.1.1 - Formats

The task force evaluated a set of digital essence formats, among them D1, D2, DV, DV-based, MPEG-2, Betacam SX, Digital Betacam and Digital-S.

The outcome of the tests was that the preferred formats for editing are DV and MPEG-2 at rates around 50 Mbit/s. The reason for so high a bit rate is to minimize the degradation caused by multiple generations, so common on editing environments.

2.1.2 - Metadata

The EBU/SMpte report probably marked the dawn of the metadata age. The reported clearly defined the concepts of Content, Essence and Metadata, and the relationship Content = Essence + Metadata.

2.1.3 - Wrappers

The concept of wrappers is also very important and was the precursor of work that is currently being done in some standards bodies.

A wrapper is something (in this case, information) that binds essence and metadata, provides information for their identification, and includes additional data like temporal relationships, transitions, effects, etc.

There are already a number of wrappers available on the industry, although few of them are general purpose and non-proprietary. There is currently a trend to define the ultimate
wrapper format that will allow the free exchange of content between applications and corporations.

2.1.4 - Object-Based Control

One of the major breakthroughs of the report, and the one of utmost importance for this work, was the recognition of object-based control as the way to go on the new television studios.

Although there has been an increasing influence form the IT industry on the television world, the control systems are still very primary. The task force envisioned, as already do some companies and organizations, that it would be of great advantage if object-oriented distributed control techniques could be borrowed from the software world.
2.2 - Distributed Object-Oriented Computing

During the final years of the last decade a trend began to develop and spread the use of object-oriented software design methodologies and programming languages. This tendency still continues and it is taken for granted that object-orientation leads to the best results. Actually, the most popular general-purpose languages are object oriented and examples are easy to find, such as C++, Java and Smalltalk.

Inevitably, this trend influenced distributed computing and, from the early days of remote procedure calls (RPC) we evolved to well-developed and powerful environments such as CORBA [20], DCOM [30] and Java RMI [31].

These technologies allow applications to communicate with one another without being aware of their location. A client can transparently invoke a method on a server object, which can be on the same machine or on a remote machine on the network.

This ability, known as location transparency, is achieved by interposing an extra layer between clients and servers. This layer, generally called middleware, intercepts the call, finds the object to which the request is directed, invokes the method passing the required parameters and returns the results to the client.

Bearing in mind that a modern television studio, composed ever more of computers and computer-based devices, is by any means no more than a distributed computing system, we can easily conclude that technologies such as the ones mentioned before can be of great advantage.

At the early stages of the work described in this dissertation, a choice had to be made with regard to which of the available alternatives to use. The next sections describe succinctly each one of them and then delve into the one that was considered the best fit for the purpose.

For a more thorough comparison, please refer to [45].
2.2.1 - DCOM vs. Java vs. CORBA

DCOM, or Distributed COM\(^1\), is a distributed computing technology developed by Microsoft Corporation. DCOM is based on COM, the Microsoft’s component model, but was extended in order to support distributed computing.

In spite of being a powerful technology that performs very well under some circumstances, DCOM has several inherent problems.

Firstly, it is a proprietary technology, which means that Microsoft is the sole decider of its future. History has it that this kind of state of affairs can lead to unpleasant surprises in the near future.

Secondly, however very well supported in the Windows environment, there is (almost) no support for other systems. Of course, this obviates any hope of using it on heterogeneous environments.

As well as DCOM, the Java Remote Method Invocation (RMI) technology suffers from the fact of being a proprietary technology. Though slightly more open and better specified than its counterpart from Microsoft, RMI is part of the Java Software Development Kit (SDK), a product developed and maintained by Sun Microsystems Inc.

Besides that, RMI only works within Java environments. Of course, Java can run on any system for which there is a Java Virtual Machine (and most do have), but nevertheless this fact limits its flexibility.

Java RMI does have some significant strengths, one of them drawing exactly from the fact that it only works on Java environments: the ability to pass not only the state of an object, but also the behaviour, by transferring code from a server to a client application.

Moreover, Sun is working on interoperability with other platforms, such as CORBA, by using IIOP (see below) as the communications protocol for Java RMI.

Finally, the other major contender is the Common Object Request Broker Architecture (CORBA).

CORBA is an industry standard, developed and enhanced since the early nineties by the Object Management Group, a not-for-profit consortium with over 800 members that develops specifications for object-oriented systems.

\(^1\) Microsoft used to call COM the Component Object Model. However, they recently proclaimed COM as just being a name with no special meaning.
The main strength of CORBA, and the OMG specifications in general, is its openness. CORBA is completely free and developed by an open membership association, assuring its users that unilateral decisions will not be taken.

Also of utmost importance is the ability of CORBA to run in any platform and to be used with conjunction with almost every major programming language.

Besides that, the OMG also specified a set of extremely useful services, known collectively as Common Object Services (COS), that simplify the development of systems by making available implementations of commonly needed features.

For all these reasons, CORBA was selected for implementing our work and a detailed explanation of CORBA is presented next.

### 2.2.2 - CORBA Overview

The work of the OMG began with the definition of the Object Management Architecture, or OMA, that stipulated the conceptual basis for the subsequent development of CORBA.

The first version of CORBA appeared in 1991 and was followed by several enhancements until the current version, CORBA 2.3.1. Every one of those versions tried to enhance CORBA’s most important strengths: language independence, platform independence and interoperability.

CORBA is built around three basic blocks: the OMG Interface Definition Language (IDL), the Object Request Broker (ORB) and the Generic Inter-ORB Protocol (GIOP).

### 2.2.2.1 - The IDL

IDL is a neutral intermediate language that is used to specify an object’s interface with any potential client. The CORBA IDL is a declarative language, which separates interfaces from implementation details. This means that IDL only specifies the interface an object exposes to the clients and never imposes any kind of implementation detail.

The IDL grammar is a subset of C++ with a few additional keywords and can be used to completely specify class interfaces, namely the parent classes it inherits from, the exceptions it raises and the methods it supports, including the input and output parameters and their data types.
After having specified the interface of an object, the CORBA programmer invokes a
compiler to generate code that will interface the implementation of the object with the
ORB. The OMG published mappings from IDL to several general-purpose languages
(among them C, C++, Java and Smalltalk) that define how the IDL will be translated to any
one of those languages.

The IDL is the key to CORBA interoperability because it defines an object’s interface very
strictly. However, nothing is said about the implementation. The actual running code is
hidden behind the interface. It is encapsulated by a boundary that clients cannot cross.

![Diagram of request passing from a client to an object implementation]

**Figure 2-1 - A request passing from a client to an object implementation**

Figure 2-1 shows the participants in a method invocation in CORBA. The stub and the
skeleton are code layers written in the same language as the object implementation. These
are generated by the IDL compiler, according to the CORBA mapping rules as defined by
the OMG.

### 2.2.2.2 - The ORB

The ORB is responsible for the management of almost everything in a CORBA interaction.

Whenever a client invokes a method on a server, the stub code passes the request to the
ORB. Of course, the ORB must find out how to convey the request to the destination.

Every CORBA object is identified by a handle known as Interoperable Object Reference
(IOR). An IOR, which is completely opaque to the client, contains information like the
interface type of the target object, the protocols that can be used to contact with the ORB
where it is running, its network address and the key that uniquely identifies it on the
ORB’s object space.
The ORB uses this information to set up a network connection with the target ORB and then uses a standard protocol (see below) to send the request.

Before the request is sent, the ORB, with the assistance of the stub code, marshals the parameters of the method and some additional information into a packet of data.

On the other end of the connection, the target ORB cooperates with the skeleton code to de-marshal the information and to hand out the request to the implementation object. When the method returns, the return values follow the same process back to the client object.

The ORB also performs some additional chores, including object activation, de-multiplexing of requests, concurrency management, etc.

2.2.2.3 - GIOP

The GIOP is the basic inter-ORB language and defines a set of rules that are made concrete on actual protocols. ORBs can use any one of the GIOP derivatives to communicate, including specialised Environment-Specific Inter-ORB Protocols (ESIOP). However, they all must support the basic IIOP, or Internet Inter-ORB Protocol.

This protocol defines how information is passed between ORBs using TCP/IP connections and how TCP/IP address information is represented on an IOR.

![Figure 2-2 - CORBA local and remote method invocation](image)

It is important to note that an ORB is allowed to optimise invocations between objects residing within the same ORB. This can be done, for instance, by using direct method calls instead of the network loop-back connection.

Figure 2-1 shows a request being invoked within the same ORB and between two different ORBs.
2.2.2.4 - Common Object Services

The OMG defined the interfaces for a set of services that are useful for several applications. This fostered the emergence of several implementations, realizing the full potential of CORBA as an excellent platform for the development of interoperable applications.

The most relevant of these services are presented below.

2.2.2.4.1 - Naming Service

The Naming Service is basically a repository of CORBA object references. Server objects publish their references by registering with the naming service under a certain name. Later on, clients can browse the service and find out the ones they are interested in.

The naming service is organized as a directed graph, possibly cyclic, where nodes are called Naming Contexts and leaves are CORBA references. The use of naming contexts is useful for a better organization of the namespace, displaying a tidier view than of a flat namespace.

2.2.2.4.2 - Trading Service

The Trading Service is similar to the Naming Service in that it also acts as a repository of CORBA references. However, names are not used to identify references, but rather properties are used to search for objects.

When a server object wants to publish itself, it sends the Trading Service a service offer. This offer contains a set of properties of the target object, properties that can either be static or dynamic.

When a client wants to locate a service that provides a certain value, it queries the service which in turn will return objects that satisfy the desired constraints.

For this purpose, the Trading Service specification defines a powerful Trader Constraint Language that is used to indicate the required property values or ranges.

The Trading Service is of utmost value in environments where clients mostly want to search services rather than specific objects.

2.2.2.4.3 - Event and Notification Services

The Event Service defines the interface for event channels that can be used to asynchronously convey events from event producers to event consumers.
Objects that produce events register on the event channel in order to send events to interested consumers; on the other hand, event consumers indicate that they want to receive events.

The format of the events is completely general and two modes of operation are supported for each end of the channel: push and pull modes.

In push mode, producers invoke the channel whenever they generate an event. Similarly, the channel notifies the consumers whenever a new event is available.

In pull mode, the channel polls the producers for new events, the same happening on the other end, where consumers poll the channel for more events.

The Notification Service is a newer specification and is a superset of the Event Service with significant additions, the most important being the ability to specify filters on every end of the channel. This way, for instance, consumers can specify which events they are interested in, reducing the overhead caused by unimportant events.

These services show their full capability in environments where the relationships between objects are quite loose. The decoupling between producers and consumers allows the object topology to be dynamic and even be specified during the deployment phase.

2.2.2.4.4 - Property Service

The Property Service defines an interface for objects that want to expose properties in a dynamic fashion.

This allows for the writing of applications that browse object's properties in a consistent way, even if they have completely disparate interfaces. An important application would be the creation of user interfaces for setting or viewing properties of objects without knowing their types.

2.2.2.4.5 - Security Service

The Security Service defines a model for authenticating users and securely passing identification information between objects. It supports several encryption and authentication protocols and is completely transparent to the programmer, in the sense that the IDL definitions do not have to be changed in order for this additional information to be sent.
Its interest is obvious in environments where some kind of security is necessary and was a major improvement to CORBA.

2.2.2.4.6 - Transaction Service
The Transaction Service has mechanisms for the management of transactions in a distributed CORBA environment.

Transaction information is passed transparently between different objects, permitting common operations like committing or rolling back a transaction.

2.2.2.4.7 - Audio/Video Streams Service
This service is not part of the generic COS framework but rather part of the CORBA Telecommunications Specifications. Although very simple, it defines methods for the control of media streams and their transfer between distributed clients of media servers.

2.2.2.5 - Recent Additions
The latest versions of CORBA unveiled a set of changes or additions to the previous specifications, some of which are worth noting.

One of the major advances was the specification of the Portable Object Adapter, or POA. The POA, which supersedes the previous Basic Object Adapter (BOA), defines a whole new relationship between the ORB and the implementation objects.

The BOA suffered from a certain under specification, which led ORB implementers to adopt different approaches to solve latent issues. This obviously led to huge differences between code written for different ORBs and to the consequent difficulty in porting code from one implementation to another. The POA unified all these approaches, making the porting process almost trivial.

In addition, the POA defines an all-new set of features, among which are an almost unlimited flexibility on the mapping between CORBA objects and their implementations.

A good example is the use of ServantLocators and ServantActivators that make it possible to have one single object instance (or servant) receiving requests on behalf of any number of CORBA objects.

This improves CORBA scalability by allowing, for instance, the existence of one CORBA reference per row in a database and, upon invocation, the sole servant will find out which one the client is referring to.
Another addition is the Asynchronous Method Invocation (AMI). This new model relaxes the semantics of CORBA invocations, which previously were necessarily synchronous from the client point of view. For example, it is now possible to invoke a method, have it return immediately and later on poll the ORB for the result of the invocation.

Of greatest importance for specialist embedded hardware are two new CORBA standards: the Real-Time CORBA (RTCORBA) and the minimumCORBA.

The former allows users to express real-time constraints and scheduling options, extending the scope of CORBA to time-critical applications. The latter defines a minimum subset of a CORBA-compliant ORB in order to ease the use of CORBA in memory-constrained devices.

2.2.2.6 - Available implementations

The number of existing CORBA applications is increasingly vast, supporting mainstream languages from C++ and Java, to less used languages like Ada and passing by scripting environments like TCL or Python.

Almost all operating systems are supported and a host of specialised implementations exist for specific areas like real-time systems.

Important is also the availability of many free, or at least open-source and free for non-commercial use, implementations with very good quality and excellent support, which gives almost everyone the ability to use CORBA.
2.3 - Standardisation Work

2.3.1 - SMPTE

The Society of Motion Picture and Television Engineers is the most active standards body in the broadcasting world. It has produced a host of recommendations and engineering guidelines that help regulate the world of television.

Two of the most relevant tasks recently undertaken by the SMPTE are described in the next sections.

2.3.1.1 - Universal Material Identifier

The Universal Material Identifier, or UMID, is a proposed standard, currently under definition within the SMPTE, which will allow the unambiguous identification of any piece of essence, whether it is an audio or video clip, a still image, or any other kind of material.

Quoting from the proposed specification, a UMID "(...) allows the unique identification of all material so that it can always be recognised whether streamed or stored locally or remotely or archived."

UMIDs are globally unique labels that will be embedded on any piece of essence and will accompany it throughout its lifetime, providing unambiguous identification of every item.

![Figure 2-3 - UMID Format](image)

The UMID, whose structure is shown in Figure 2-3, is defined in two forms: the basic and the extended UMID. The basic UMID must always be used and identifies unambiguously any clip of material, whereas the extended UMID provides additional metadata so that every content unit inside a clip can be identified\(^2\).

\(^2\) Content unit is defined as the quantum of duration of material. Clip is defined as an integer number of content units.
Four fields compose the basic UMID: a SMTE Universal Label that identifies it as being an UMID, a length field, an instance number and a material number.

The instance and material numbers provide the uniqueness to every UMID and [22] specifies recommended practices for its generation.

In short, material will be allocated a material number on a shot-by-shot basis, which will remain immutable throughout its lifetime. The instance number must be 1 at the time of generation but will be updated for every copy of material.

It is important to distinguish simple copies of material, possibly with different resolutions, which will keep the same material number, from any kind of transformation, such as editing or colour adjustment, in which case the material number must be changed.

The proposed standard defines methods for the automatic generation of both material and instance numbers, that are guaranteed to be unique with very high probability. This is of greatest importance since it allows UMIDs to be generated without the need of a central authority.

The UMID is expected to be the major element in the linking between Technical, Production, Archiving, Business and Service processes in an organisation, allowing the bi-directional cross-referencing between essence and metadata.

2.3.1.2 - Advanced System Control Architecture

The Advanced System Control Architecture (ASCA) is being defined within the Working Group on Advanced System Control Architecture, part of the SMPTE S22 group.

It was formed as a consequence of the results of the EBU/SMPTE Task Force and its aim is to unify the approaches presented by the different manufacturers of control and monitoring equipment for studio equipment.

It is expected that the work of this group will eventually lead to the definition of a standard on studio control and management, provided that there is enough consensus on the industry and that this group achieves critical mass. However, the progress has been extremely slow so far and it is possible that an eventual standard will only be released too late for the industry to wait.
The only concrete outcome of this group so far is the “System Overview” document [24]. This document provides an overview of the ASCA and will try to identify specific areas for SMPTE standardisation efforts.

![Figure 2-4 - SMPTE ASCA Component Diagram (source: [24])](image)

According to this document, a successful architecture must be:

1. Flexible enough to accommodate varying enabling technologies where possible;
2. Independent of specific technologies where possible;
3. Scalable to a range of platforms and environments;
4. Extensible to adapt to emerging technologies;
5. Modular in nature allowing functional pieces of the system to be used as building blocks;
6. Offer a viable migration path for existing facilities.

Perhaps the most important achievement of this document is the consensus on a layered view of the system, which is also sliced in different functional planes. This view is presented in Figure 2-4.
The planes normalize access to the primary studio assets, namely content, network, and devices and a usage abstraction from content through to device is provided to support varying levels of workflow within the studio (Figure 2-5).

![Diagram](image_url)

**Figure 2-5 - SMPTE ASCA Plane Usage Abstraction (source: [24])**

The layers provide access to the reusable system functions provided by the studio devices and resources. Layers also cater for physical device and network independence where possible, in order to allow for the highest degree of application portability (Figure 2-6).

![Diagram](image_url)

**Figure 2-6 - SMPTE ASCA Functional Planes View (source: [24])**

The ASCA working group intends to use a distributed object-based platform as the underlying communications system. However, the group has not yet achieved consensus on which of the available platforms to use and thus how interfaces between components
will be defined. Among the contenders are CORBA [20], XML-based systems such as SOAP [48], DCOM [30] and Jini [37].

Meanwhile, the group has agreed that the UML [23] will be used to define the system structure and interactions, postponing the commitment to a particular platform to a later stage.

2.3.2 - Pro-MPEG Forum

The Pro-MPEG Forum is an association of broadcasters and programme makers, equipment manufacturers and component suppliers who are in some way involved with the use of MPEG-2.

The main focus of the Forum is to promote interoperability of MPEG-2 equipment and applications on the professional television programme making and broadcasting environment, according to the requirements of broadcasters and other end-users.

To achieve this, Pro-MPEG proposes guidelines and codes of practice to augment MPEG-2 standards and accelerate their implementation. The Forum is also proactive in fostering interoperability with other compression methods.

The achievements of Pro-MPEG are regularly shown on trade shows and other public demonstrations.

The Pro-MPEG Forum is divided in several specialised working groups. At present, the active groups are File Interchange, Operating Ranges, ATM Interoperability, User Requirements and Networking & Control.

The first and the last groups are of most relevance to the work described in this text.

The File Interchange group is currently defining, along with the Advanced Authoring Format Association (AAF) [39], the Media Exchange Format (MXF).

The MXF is a wrapper format that most probably will be widely used in the industry for the exchange of Content within and between organisations.

The Networking & Control group is organising demonstrations of interoperability between distributed object-oriented control systems and devices. It is also starting a crusade to evangelise the industry on the best ways to use the tools and techniques developed on the IT world.
The work described in this text was presented as input to this group and is serving as part of the baseline for its work.

2.3.3 - DSM-CC

2.3.3.1 - Overview

DSM-CC (Digital Storage Media - Command and Control) is an ISO/IEC standard [26], part of the full MPEG-2 specification, which was developed for the delivery of multimedia broadband services. The motivation for the development of this standard stemmed from the need of a consensual protocol that allowed the widespread deployment of such services.

The Digital Audio-Visual Council (DAVIC) [28] was one of the first organisations to recommend the use of DSM-CC. DAVIC encourages the success of emerging digital audio-visual applications and services over various networks by the timely availability of internationally agreed specifications of open interfaces and protocols. DAVIC has adopted DSM-CC in its specification [28] where it is used as the protocol for control of multimedia interactive sessions as well as the resources within the sessions, and for service level interactions.

DSM-CC is transport layer independent, which means that applications written to use DSM-CC needs not to concern with the underlying transport layer used between the server and the client. The goal is to allow the same application to be delivered over a multitude of broadband networks, ranging from a pure MPEG-2 Transport Stream network, to a Core ATM and a variety of access networks, with or without ATM, even including high-speed local area networks (LANs), to end-to-end ATM networks.

DSM-CC is defined in terms of a simple functional reference model, shown in Figure 2-7. The model shows client and server entities (jointly called users) that use a network to communicate with each other.

Clients are, in general, devices such as set top boxes that consume multi-media content. Servers are entities that provide multi-media content and services. A server may be a distributed system, contain multiple computing platforms and thus multiple connection points to the network.
The reference model shows User-to-User (U-U) connections, alongside with User-to-Network connections (U-N), where U-U connections are between client and server, whereas U-N involve a user and the network.

The model also shows the network as containing a Session and Resource Manager (SRM). This entity is used to manage all the resources on the network and can police the users based on policies set when resources were requested.

The Service Gateway is the entry to the system and gives the user a view of the available services.

DSM-CC defines exactly the format of U-N messages but U-U messages are left to agreement between client and server. Nevertheless, a set of generic services is defined and a Remote Procedure Call (RPC) protocol is used to perform the invocation of these services.

A fundamental concept in DSM-CC is the one of a session, defined as an association between two users, providing the capability to group together the resources needed for the service. For example, when a client accesses a service and sets-up a session with a server, all the resources associated with it will be released when the session is finished. Besides that, the grouping of resources is also useful for operations like billing and administration.

Typical resources acquired during a session include U-U connections and data connections, such as the ones used to transfer MPEG-2 streams.

One of the advantages of DSM-CC is its ability to set-up end-to-end connections through disparate networks. This ability is extremely useful in applications such as video-on-demand, where different kinds of networks are used for backbones and end-user access.

DSM-CC also grants support for automatic configuration of clients. When a client connects to the network, it can ask the network to configure it, including even the ability to download software to the client. The mechanism used for the downloading is the DSM-CC Data Carousel, which was later adopted in DVB for the downloading of de-scrambling codes.

The U-U interface definitions are written using the Interface Definition Language (IDL) and provide two distinct interfaces: an Application Portability Interface (API), defined for programmers writing applications that run on clients, and a Service Interoperability Interface (SII), defined to allow interoperation of clients and servers from different
manufacturers. DSM-CC does not specify which RPC scheme is used but recommends the use of CORBA 2.0.

Several U-U interfaces are defined in DSM-CC but probably the most important are Directory and Stream. Directory allows a client to list the streams available in a server, while Stream lets the user control the video stream with commands similar to the ones usually available on a VTR.
This chapter presents research projects that actually implemented solutions along the guidelines presented in the previous chapter.

The first two sections present projects on which the author took part and that directly led to the work described in this text.

The last section describes a similar project undertaken by other companies in this area from which useful comparisons can be drawn.
3.1 - ATLANTIC

3.1.1 - Introduction

Having had its beginning in 1995, the ATLANTIC project [11] was a research effort aiming at the development of technology for the implementation of a TV production studio totally based on MPEG-2. Lasting more than three years, the project was run by a consortium integrating BBC R&D, CSELT, EPFL, ENST, FhG, INESC and Snell & Wilcox, with funding granted by the European Community, under the ACTS framework.

Among the main goals of ATLANTIC was the demonstration of the feasibility of performing the editing of TV programmes in MPEG-2 format with quality and cost effectiveness. The project developed new equipment and techniques in order to accomplish this goal, which overcame the threat to the technical quality of the television sound and picture signals that would normally result from the use of bit-rate reduction, and in particular from decoding to PCM form and recoding. This was achieved by avoiding decoding/recoding operations wherever possible and by reusing the original coding decisions, which were conveyed to the point of recoding by MOLE signals.

Another objective of ATLANTIC was to prove the possibility of using inexpensive computers to perform tasks usually done by specialist equipment. In order to do it, a news studio was developed composed of various pieces of equipment interconnected via an ATM network, exchanging all kinds of information between the different computers.

ATM was chosen for its unique characteristics, such as flexibility, scalability, provision of bandwidth on demand and the support of a wide range of quality of service requirements. Nonetheless, the need to provide reliable transmission of MPEG-2 streams within the studio led to the selection of TCP as the transport mechanism of choice, running over ATM Adaptation Layer 5 (AAL 5).

3.1.2 - Reference Model

The various components of the ATLANTIC post-production facility are shown in the reference model of this studio (Figure 3-1).

The Format Converter is the gateway of the ATLANTIC studio, interfacing it to the outside world and receiving all incoming essence in MPEG-2 Single Program Transport
Stream Format. The Format Converter then recovers all elementary components of the stream and stores them as separate Packetized Elementary Streams (PES) in the Server.

![Diagram of the ATLANTIC post-production facility reference model](source: [11])

The Server stores all essence files along with index files that facilitate the conversion of timestamps into byte offsets, thus making it possible the time-based random access into the streams.

The studio infrastructure also allows the user to edit video sequences with single frame accuracy; thus, instead of working with long-GOP MPEG2 video streams, a MPEG-1 I-frame only version of each stream is created by the Browse Track Generator, with the purpose of being used in rough-editing operations.

The Journalist Editing Workstation (JWS) is an editing facility for post-production, consisting of a PC with a GUI and MPEG-2 decoder boards.

The JWS lets users produce their own programmes by creating EDLs that in turn will be passed to the Edit Conformer, which is responsible for using the EDL and the MPEG-2 streams in order to generate the final programme and store it as a MPEG Transport Stream in the Finished Programme Server.

### 3.1.3 - Application Control

Having various components interconnected via a network, a need obviously arose to define a suitable means for controlling and interconnecting the different pieces of software.
As an initial approach, the usual tools available in most operating systems were tried, namely the use of remote logins for controlling the different processes and the employment of the Network File System (NFS) and the File Transfer Protocol (FTP) for data transfer.

Of course, the use of these tools revealed a non-integrated and difficult to manage system and the need for an integrated command and control plus data transfer architecture became obvious.

Additionally, the performance of methods like NFS is very poor and there was a requirement to optimise data transfers over different media, an example being the direct use of AAL 5.

It was decided then to evaluate the ability of DSM-CC to perform such a task. The choice of DSM-CC arose from its inception as a command and control framework, which also supported data transfer over various media.

An initial implementation of a subset of DSM-CC was quite successful, giving the JWS the ability to control the Main Server.

Unfortunately, DSM-CC started revealing its inadequacy to this purpose and it was decided to develop a new architecture to address this problem.

The first iteration produced a solution named Studio Digital Storage Media (SDSM), which was based on DSM-CC but simpler and more appropriate to the situation in hand.

With the insight gained from these experiences, it was decided to start from a blank sheet and design a completely new system, more fit for this purpose. The result was named Distributed Middleware for Multimedia command and Control, or DIMICC, and is the focus of this dissertation.
3.2 - ORBIT

3.2.1 - Introduction

After the success of ATLANTIC, some of the participants in the consortium thought that there was still work to be done in order to consolidate some of the developments made in the project. Namely, it was thought that a trial implementation, that showed enough reliability and that could satisfy the immediate needs of the staff that would be using it, was necessary in order to be accepted in an operational broadcasting environment.

With this goal in mind, BBC R&D and INESC Porto decided to launch the Object Re-configurable Broadcasting using IT (ORBiT) project.

ORBiT is intended to provide, in a timescale of two years, a pilot implementation based upon ATLANTIC. The pilot will be a small-scale model capable of handling "live" and recorded signals, from local and distant sources, and of demonstrating the facilities and interconnections that will be needed in a full-scale operation. The pilot will also provide the means for operational staff to use the system to gain experience and to feed back their comments and criticisms.

The main objectives of ORBIT are to identify best practice in the following areas: migration of content handling away from specialist broadcast equipment towards economical mass-market IT hardware; integration of media asset management and content handling tools; suitability of architecture for use on a corporate-wide scale; ready access to content at the desktop; automation of the essence and metadata handling process; reconfigurability for different content formats, including any necessary conversion; and reconfigurability to cope with various production processes and programme genres.

Moreover, ORBIT will pursue standardisation of the techniques developed in ATLANTIC and proclaim the advantages of using middleware in order to maximize flexibility and scalability of television facilities.

3.2.2 - Reference Model

In order to be able to advise on the mentioned issues, the project is developing several interconnected networks, each corresponding to a single production, archive or play-out area. Figure 3-2, which shows one production area connected to other similar areas via gateways and an inter-area network, depicts the various process stages involved:
- The intake host captures essence at browse and full quality, from sources such as VTRs, cameras or live feeds. The acquisition and the source itself are controlled using a GUI on a client application, which is also capable of monitoring the input;

- In order to allow searching and browsing, the local or remote metadata databases can be queried, providing the user with a list of relevant material and thumbnails. Material can then be previewed at browse quality on the workstation, with the usual trick modes available;

![Diagram of ORBIT reference model](image)

*Figure 3-2 - The ORBIT reference model (source: [13])*

- Simple edits can also be compiled at browse resolution on the workstation, and the resulting edit list used by a server to conform the full quality material in order to generate the final programme. For some type of programme, however, this EDL can be used as a “rough-cut” input to a more sophisticated on-line editor;

- The quality monitoring workstation, naturally, permits the quality assessment of programmes, which are streamed from the content servers and decoded to broadcast quality monitor and loudspeakers.
3.3 - DS-CC

3.3.1 - Introduction

The Digital Studio Command and Control specification [48] is the outcome of the HDTV Broadcast Collaboration, a project funded by the American National Institute for Standards and Technology Advanced Technology Programme (NIST ATP). Several organisations took part on this project, namely, Sarnoff Corporation, Thomson, IBM, Philips, Comark, Modular, SUN and MCI.

This project, which is contemporary of ATLANTIC, played a very important role as one of the precursors of the current work towards standardisation, serving as a baseline for efforts like the SMPTE ASCA.

DS-CC, which was heavily influenced by DSM-CC, defines a control architecture and application programmer interfaces (API) for a significant set of studio components that comprise an advanced digital studio. These components are suitable for managing audio, video and data communications within and between studios and issues like session management, real-time control, proxy device control and stream management are addressed by this architecture.

3.3.2 - Studio Control Overview

For studio control, DS-CC prescribes the use of a network of distributed objects communicating via CORBA. These objects are instances of classes defined in the document and provide means of controlling the physical devices they represent.

DS-CC broadly defines three groups of classes: device resources, streamable resources and software resources. Examples for each one are, respectively, media servers, encoders, switchers and cameras; streams; studio play-to-air managers.

The object interfaces are defined syntactically using the CORBA Interface Definition Language (IDL) and semantically using textual descriptions. As usual in CORBA systems, nothing is said with regard to the implementation.

DS-CC is designed to support the easy creation and control of media streams across different studio devices. It also supports several system level services that can be used to access content and other facilities and to provide for optimum utilization of scarce resources like the network.
Therefore, the object model is divided into two levels: the service model and the streaming model.

### 3.3.3 - System Model

The system model contains a set of services that both control the whole system and give client access to the available resources.

The most important entity of the system level services is the System Resource Manager (SRM). The SRM is composed of three separate sub-systems that perform basic system-wide functions: resource registration and discovery, network bandwidth reservation and allocation, and device and device port reservation and allocation.

#### 3.3.3.1 - Device and Tool Registration

One of the sub-systems of the SRM is the Service Gateway (SG), the central repository of all resources available in the studio.

Upon installation, devices and tools are required to register with the SG, in order for clients to be able to use their services. The registration process gives the system the ability to track all studio resources, which is fundamental for resource allocation and reservation.

Whenever a client desires to use some kind of resource, it first must be able to locate it in the SG, where a structured view of the resources is provided. The client can thus issue a request for a particular device when its name is known in advance, or can select one among the available devices of the required category.

#### 3.3.3.2 - Resource Reservation and Allocation

DS-CC has mechanisms for reservation of resources prior to their use. This ability is fundamental in order to ensure that critical operation will have the necessary resources to perform satisfactorily. Just before the resources will actually be needed, the client is required to commit the reservation in order for the resources to be available.

DS-CC supports three scenarios for resource allocation and reservation:

**Case I:** SRM handles reservation; SRM allocates device resources and network bandwidth.
In this scenario, the SRM has complete control over the studio resources and over the network fabric. This is probably the ideal situation in terms of control but may not be implementable under certain circumstances. This scenario is called SRM centric.

**Case II: SRM handles reservation; SRM allocates device resources; Stream Node negotiates connection with the network.**

In such a system, the SRM still manages the resources but has no direct control over the actual connection management. This scenario is called SRM enabled.

**Case III: No reservation; Client allocates device resources; Stream Node negotiates connection with the network.**

In this scenario no reservation of resource prior to their use is possible, so critical operations cannot be guaranteed in advance. In reality, this would be a “best effort” mechanism and is not favoured by the promoters of DS-CC.

### 3.3.4 - Streaming Model

The most important classes in the streaming model are *Device*, *DevicePort*, *StreamNode* and *Stream*.

The *Device* class, as the name suggests, contains all the hardware level attributes and control functions. Every one of these has as many *DevicePort* objects as the existing physical ports. *DevicePort* objects are mainly used during the reservation process in order to estimate the capacity of the involved ports.

Objects of the *StreamNode* class control the stream creation and set-up and represent data to be streamed, whereas *Stream* objects represent the control points for the flow of data.

Whenever a client wants to set-up a connection between two devices, it requests the SRM to create it and gets back a *Stream* object. This object can then be used to control the flow of data between the devices.

### 3.3.5 - Factory Objects

Within DS-CC, all studio components have an associated factory object. The underlying reason is that, when several clients obtain a reference to an object via the SRM, they all get a reference to the same object.
To separate the contexts of the different sessions, it is required to request the creation of a new object, with the same interface, before it can be used.

### 3.3.6 - Events

It is common in an environment such as a television studio to exist the need for monitoring the state of a device. Examples include an operator console that monitors the state of all devices or a client monitoring devices and streams associated with a play-out request.

In order to implement this kind of functionality, DS-CC uses events and event channels. Event channels act as conduits of events, asynchronously sending events from producers to consumers.

This approach promotes the decoupling of observer and observed objects, a pattern usually known as Observer [2], simplifying the implementation of the monitored objects and allowing the dynamic addition of monitoring tools.

DS-CC specifies the interface for two kinds of event channels: basic and extended event channels.

### 3.3.7 - Studio Components

#### 3.3.7.1 - Content Servers

Content Servers are devices that are able to store and playback essence on request. Content Servers expose a control interface that allows basic control operations and also have requirements to manage content stored on fixed or removable media.

Contrasting with devices that do not support stored content, Content Servers do not maintain static StreamNode objects. Rather, they create StreamingElement (a subclass of StreamNode with content-related methods) objects on demand, and are thus factories for StreamingElement objects.

Content Servers are sub-divided into Media Servers, Tape Devices and Tape Servers.

Media Servers have methods for finding content stored within them, using a key that uniquely identifies the piece of content within that server. When looking for a particular piece of content, clients will usually refer to system services in order to get the server name and content key.
Tape Devices represent VTRs or similar devices, whereas Tape Servers control several Tape Devices.

3.3.7.2 - Other Components

DS-CC defines several other component types, including Encoders, Transcoders, Switchers and Non-linear editors.

3.3.8 - Media and Metadata Management

DS-CC defines a hierarchical model for content management and has support for manual and automated metadata annotation.

The main components in this subsystem are the Library Server, the Browse Stations and the Annotation Stations.

The Library Server is the central entity that is knowledgeable about all the content in the studio. It is thus responsible for contacting the media servers and retrieving all the metadata about the existing clips.

Additionally, the Library Server stores additional invariant metadata, such as the one created by manual or automated annotation.

Finally, it is responsible for assisting in the management of collections of removable containers, such as cassettes, cartridges, reels, etc.

The Library Server interacts with the rest of the system both by allowing clients to query it in order to find content and by locating all Media Servers, with the assistance of the SRM, in order to retrieve metadata about their contents.

The Browse Stations are clients of the Library Server specialised in interrogating it and preview the contents.

Annotation Stations are servers, possibly unattended, which process essence in order to extract metadata and pass that information to the Library Server.

3.3.9 - Conclusion

DS-CC is a very well crafted document, result of a long study of the necessities of a modern digital television studio and of the application of the new technologies developed within the IT world.
However, at least subjectively, some flaws can be identified in this specification, either due to deliberate choices of the designers or caused by the evolution of the technology, which inevitably obsoletes some decisions.

One of those decisions is not to use any of the services defined in the CORBA Common Object Services (COS) specification [27]. Although the argument that DS-CC would be vulnerable to potential changes in those standards can be considered reasonable, the use of already available standards would benefit both users and implementers of DS-CC based systems.

Potentially interesting services, among others, would be the Naming and/or Trading Services for the registration and discovery of resources on the Service Gateway; the Event Service (or the new Notification Service, which was not yet defined at the time) as a replacement for the DS-CC Event Channels; and the Security Service as a more robust approach to the security problem.

Another point where DS-CC reveals a design weakness is on the inconsistent use of factories. When an object acts as a factory for other objects and their users are required to request a new object before they can be used, it is fundamental that the class of the factory be different from the class of the product objects.

The reason for this requirement can be simply explained: when a client gains access to a factory object and, instead of requesting the creation of a new object, tries to use that object directly, the only way the server can flag the error is by raising a run time exception. This error will only be reported at run time, which means that if the code is not thoroughly tested and inspected (that is, in a great percentage of most projects...) it will not be detected until it is too late.

If the class of the factory were different from the class of the product objects, the error would have been detected at compile time, eradicating a very troublesome cause of headaches.

In DS-CC, there is an example of a well-crafted factory pattern, specifically the one on which SRM and SRMSession take part. Unfortunately, all the others present the opposite and undesirable behaviour.

Similar problems of poor object-oriented design and type safety are spread along the specification. One of them is the use of close methods on the factories rather than on the products, of which an example is the SRM/SRMSession relationship.
Another one is the use of non-opaque keys to access streams via the media servers, rather than use CORBA references to access them directly. The reason for this choice is twofold: firstly, it seems that the designers did want to have some kind of human readable information on the keys; secondly, it was thought that the use of one CORBA object per stream would be exaggerated in terms of memory requirements. However, the use of opaque keys is much more desirable because otherwise users tend to fiddle with them and break type safety. And to obviate the second reason, the latest CORBA specifications define mechanisms like ServantActivators and ServantLocators that allow the same implementation object to incarnate several CORBA objects, minimizing the overhead imposed by such a cleaner design.

Another point where DS-CC seems to lack some polishing is the presence of some “garbage” on the specification, most of which are remains from the DSM-CC influence. There are still some definitions that are not used at all and others that could be simplified through careful revision.

Finally, it seems that parts of the system are too specific for the environment where it was developed, and thus probably overly complex, whereas some areas would need to be better defined, namely the connection establishment procedures and the use of transport protocols.
During the lifetime of the ACTS ATLANTIC project, the need for a command and control architecture for the whole studio was deemed as absolutely necessary.

A survey of the solutions available at the time, followed by the testing of one of them, concluded that no one fulfilled all the requirements. This state of affairs led to the design and implementation of a completely new system, which was named Distributed Middleware for Multimedia Command and Control, or DIMICC.

DIMICC became the control and communication platform for the ATLANTIC and ORBIT projects and has since been continuously evolved to address the new requirements of its users.

This chapter gives an overview over the context on which DIMICC was developed, describes its initial requirements and then focus on the analysis and design phases.
4.1 - Context

The precursor of this work was the ATLANTIC project, specifically its development as an aggregate of several computer-based components interconnected via a network.

As was already mentioned, the requirements of such a distributed system led to a tentative application of DSM-CC and some immediate interesting indications. Unfortunately, several glitches started creeping in, the main problems deriving from the fact that DSM-CC is completely oriented to video on demand (VoD), thus necessarily lacking an object model that thoroughly fits a television studio and its special requirements. Just as an example, there is no provision for uploading content to a media server.

Moreover, DSM-CC is a huge specification, defining overly complex mappings for transport protocols, as well as intricate embedding of extra addressing information within CORBA object references, the latter being viewed as breaking the opacity of such references and requiring complex and non-standard interactions with the ORB.

As a result of these issues, and upon the realization that a great deal of change would have to be made to DSM-CC, it was decided to design a wholly new solution.
4.2 - Requirements

The first requirement for DIMICC was that it should be based on distributed object-oriented techniques and, from our experience with DSM-CC plus its record as an open, vendor independent, highly available technology, CORBA would be the chosen technology.

Secondly, it should be as much as possible built upon existing de jure or industry standards. Obviously, CORBA Common Object Services should be used whenever appropriate.

Thirdly, it should be based on a full-blown object model that treated as first-class objects both tangible things, such as devices, and intangible items, such as essence. Furthermore, this model should be completely extensible, capable of allowing the non-disruptive addition of new classes of objects, in order to support new devices or items.

As a fourth requirement, DIMICC should leverage CORBA abilities such as location transparency and encapsulation. Put another way, users should only care with value provided by the objects, and never, unless explicitly required, with their physical location or the way they are implemented.

Fifthly, DIMICC should promote componentisation, that is, the construction of a system based on small building blocks that could easily be tested and reused without changes on the implementation.

DIMICC should also be able to use several data transport mechanisms, in such a way that it would be transparent to its users which one was being used. Besides, there should be provision for the definition of the required quality of service (QoS) on a per-connection basis and for the use of multicast.

Seventhly, a seamless integration of metadata and essence should be achieved, always keeping the relationships and navigability between each other.

Support for management applications should also be intrinsic in the system. These would include the ability to monitor device status, network usage, alarms, etc.

Ninthly, security should be considered from the beginning of the design, even if not implemented in the initial releases.
Tenthly, it should be anticipated the need to **federate several DIMICC systems** in order to share resources in a controlled manner.

Lastly, support should exist for both **legacy and inexpensive devices** that do not have networking capabilities and for **legacy networks** such as the Serial Digital Interface (SDI).
4.3 - Analysis and Design

This section gives an overview over the major decisions taken when designing DIMICC, outlining the most important classes and presenting several UML diagrams.

The reader is asked to refer to 0 for an explanation on the notation used.

4.3.1 - Planes

The DIMICC architecture can be divided into four planes, each one being responsible for dealing with separate aspects of the whole system.

The system plane is in charge of the system-wide services that provide the users with a unified view of the whole system.

The control plane deals with issues related to controlling devices and essence, and with the reporting of important events and alarms.

The essence transfer plane takes care of the essence movement between objects, shielding users from the details of the underlying networks.

Finally, the metadata plane is accountable for the linking between metadata and essence.

4.3.2 - System plane

The system plane defines system wide services that control the whole system and gives the user the ability to manage it. The protocol stack is shown in Figure 4-1.

Central in the system is the System Naming Service (SNS), an instance of the CORBA Naming Service.

This service acts as the central repository of all system resources and is the entry point of the users into the system. When looking for a particular resource, users will browse the SNS in order to find the desired one.

Similarly, management tools can browse the service, find out which resources are available and then query them in order to gather information.

However, the preferred method for users to find resources is the System Trading Service (STS), itself an implementation of the CORBA Trading Service.
This service is registered in the SNS and users can take advantage of its ability to perform searches based on provided value, rather than on a particular name.

The third main component of this plane is the System Notification Channel (SNC), an instance of a CORBA Notification Channel also registered with the System Naming Service.

This channel conveys system-wide events from any object to listeners that may, for some reason, be interested in these events.

The main advantages of using a notification channel arise from the decoupling of producers and consumers and for the ability to install filters for every producer and/or listener.

These features simplify the implementation and non-disruptive addition of management tools. For instance, a tool can be installed when the system is up, connect itself to the SNC and monitor important events, with the rest of the system being completely unaware of this fact. Furthermore, it can install filters in order not to be disturbed by uninteresting events.

The use of a notification channel can also be practical for postponing system policies to installation time and to allow them to be changed easily. An example is the ability to have a central entity responsible for the registration and deregistration of resources on the SNS and STS. That is, instead of coding the behaviour needed to register on the repositories, with all the static and dynamic information required for that, an easily updateable script could be put in place to monitor device start-up and shutdown events and appropriately perform the required operations.
Regarding security, a system security management service is being developed, which will be available on the system naming service and will allow users to authenticate when they log into the system.

![Control Plane protocol stack diagram]

Functionality will be available for the creation of users and user groups and for associating privilege profiles with every group or user.

Using the CORBA Security Service, clients will transparently pass their credentials to servers whenever a method is invoked. The servers will then be able to query the system security service in order to retrieve the user's security profile and decide whether or not a particular action is allowed.

At present, there is no support for the federation of several DIMICC systems. However, it is envisaged that such a feature would be easily implementable using a management service responsible for passing requests and security profiles between the different systems.

### 4.3.3 - Control plane

#### 4.3.3.1 - Basic Classes

This plane is responsible for controlling resources and for providing means for the reporting of events.

One of the first requirements mentioned before is that every resource, whether tangible or intangible, should be treated as a first class object.

Thus, there are broadly two kinds of resources: those who participate in essence processing and those who do not. The former group includes devices like encoders and media items
like video streams. The latter includes, for instance, media servers, here seen just as a repository of streams, or tape libraries.

We thus have two broad classes of controllable objects, designated respectively Components and Services.

This raises the issue of ownership and locking of an object. When a client is listing the streams available on a media server, other users can concurrently use the same service. On the other hand, when a client is playing a media stream, the instantaneous state (like current position, speed, etc.) must be independent of any other user that may be simultaneously playing the same stream. The same happens, for instance, with an encoder with the addition that usually only a single user can use it at a time, and consequently locking is necessary.

![Diagram](image)

**Figure 4.3 - Component and VirtualComponent**

There is then a need to handle the concept of a session and, for that reason, we have identified a class of objects that represents a session with a component, VirtualComponent.

VirtualComponents will be transient objects created whenever a user requests a session with a Component and will represent the context of the current session of the user with that Component (Figure 4.3).

It thus seems reasonable that VirtualComponents own the logical input and outputs of the Component during a session. In this model, Sink and Source objects represent the VirtualComponent’s inputs and outputs, respectively, and are owned by the VirtualComponent. Sinks and Sources are subclasses of a generic Endpoint.
class and clients can query VirtualComponents in order to get the references for their Endpoints (Figure 4-4).

By abstracting out the concept of Endpoint, it is possible to encapsulate the complexity of connection management within these objects, hiding the user from the low level details such as the use of several network technologies and protocols.

A point worth noticing is the relationship between the concept of Component and the hardware it runs on. Clearly, most special purpose devices like encoders or VTRs have a one-to-one mapping between a Component and a hardware device. However, when computers are used to run Components, several instances of these objects can share the same device and, consequently, processing power, memory and network ports. The solution is to define a new class Device that represents the concept of actual hardware.

![Figure 4-4 - VirtualComponent, Source, Sink and Endpoint class diagram](image)

A physical device contains one or more ports that interface with the network. A good logical representation can be obtained by creating a class NetworkPort, whose instances will be owned by a Device object (Figure 4-5).

Therefore, Endpoints represent the virtual inputs and outputs of a session with a Component, that is, of a VirtualComponent, whereas NetworkPorts represent the actual interfaces with the network.
If a particular kind of network interface supports the multiplexing of several logical connections, various Endpoints can share simultaneously the same NetworkPort. Otherwise, the access will be mutually exclusive.

![Diagram](image)

**Figure 4-5 - Device, NetworkPort and Component**

### 4.3.3.1.2 - Class Specialisations

Each one of the above-mentioned classes of objects (Device, NetworkPort, Service, Component, VirtualComponent, Endpoint, plus its specialisations Source and Sink) defines abstract and common behaviour for clients dealing with the control plane.

Obviously, some of these classes will have to be specialised in order to add specific behaviour to particular kinds of resources.

Whereas Device, NetworkPort, Endpoint, Source and Sink will probably be enough for most of the applications, clearly Service, Component and VirtualComponent will have to be specialised to define appropriate interfaces for media servers, VTRs, media streams, encoders, etc.

Figure 4-6 shows as an example the specialisations needed for a VTR.

### 4.3.3.1.3 - Events and Properties

With the purpose of easing monitoring and leveraging the flexibility of the system, each of these classes will generate events and send them via a notification channel. Clients will then be able to query an object about its notification channel and to connect to it so they can listen for events.
However, and to avoid the overhead of having a notification channel per object, objects will be allowed to share notification channels and the source of the event will be identified in the event itself.

An apparent problem here is how to identify the source of events. The CORBA reference of the source object could be used, but the concept of identity is very weak in CORBA and the comparison of object references is not guaranteed to be reliable. The solution adopted is to define a hierarchical identification system where each Device is allocated a system-unique identifier. Then, each Service, Component and NetworkPort running on a Device is given an identifier unique to the namespace of the Device while VirtualComponents are uniquely identified in the context of a Component. Finally, Sources and Sinks will receive unique identifiers in the context of a VirtualComponent.

![Diagram](image)

**Figure 4-6 - Class specialisation example: the VTR package**

To exemplify, the identifier of a Sink would take the following form:

```
<device_id>/<component_id>/<virtualcomponent_id>/<sink_id>
```

Like the use of events augments flexibility, the use of properties in these objects eases the use of already available tools to view and change parameters. Additionally, it provides an elegant way for manufacturers to add specific parameters to objects and let them be configured easily.

For these reasons, it was decided that every object on this plane should derive from the PropertySet class as defined in the CORBA Property Service.
In order to factor out all the previously mentioned behaviour that is common to all objects in this plane, a class StudioObject was defined from which every other class derives.

The interface of this class supports behaviour for getting the object's id, notification channel and is itself derived from PropertySet.

4.3.3.1.4 - Non-networked Devices Support

There is a requirement to support devices that do not have network connectivity, such as legacy and inexpensive devices. Usually, these devices have RS-422 interfaces and a proprietary protocol to communicate with controllers.

In order to interface them transparently with the CORBA-based DIMICC world, the use of proxies is advocated. The typical set-up is shown in Figure 4-8, and involves one or several DIMICC Components running on a computer or similar hardware, connected both to the system network and to the device.

This special Component will then map DIMICC commands into the appropriate command in the protocol of the device. Obviously, responses or alarms sent by device will be mapped back appropriately.
4.3.4 - Essence transfer plane

The essence transfer plane, as it name suggests, is responsible for transferring essence between components and for the management of the network connections required for that task.

There is a requirement that several network technologies and protocols must be supported. Apparently, this issue should not be a problem in a CORBA environment. Unfortunately, the problem exists because CORBA will not be used for essence transfer!

Reasons for this option are various and chief among them is the poor performance of CORBA when used to stream large amounts of data, the cause for this being mostly that CORBA was not designed to optimise this use-case. To be fair, it should be noted that work is underway to obviate this problem [12], but at present it is still not a good solution.

![Diagram of Essence Transfer Plane protocol stack]

Moreover, it is a requirement that legacy networks, for which usually there are no CORBA implementations, should be usable with DIMICC and, even worse, multicast and QoS should be supported.

The solution to all this is to define a protocol for the negotiation and establishment of connections between Sources and Sinks, which, whenever instructed by a client to do so, will transparently set-up the data connections.

4.3.4.1 - Connection Management Protocol

In general, most devices only have one network interface. However, there are cases when several exist, either all connected to similar or to completely different networks. Even
when only one interface exists, several protocols may be supported on top of the data link layer.

A corollary for this is that it is possible that, when trying to set up a connection between two devices, multiple paths may be available, usually the intersection of the sets of networks and protocols supported by each of the devices, provided that full connectivity exists.

Of course, this has an effect on the way Sources and Sinks will negotiate the connection establishment.

![Connection establishment sequence diagram](image)

**Figure 4-10 - Connection establishment sequence diagram**

In DIMICC, whenever a client wants to interconnect a Source and a Sink, it will invoke a method in the Source and pass it a reference to the target Sink.

The Sink will then return a list of the protocols supported and the respective service access point (SAP) addresses. Comparing with its own list, the Source will decide which protocol to use and then establish the connection, using the appropriate signalling.

Finally, the Source will notify the Sink that the establishment has succeeded (or otherwise) and pass it a connection identifier.

Figure 4-10 shows the sequence diagram for this process.
When a connection is to be terminated, the client will instruct the Source to do so, causing the latter to inform the Sink and to release all the resources associated with the connection (see Figure 4-11).

### 4.3.4.2 - Quality of Service

A client desiring to set-up a QoS enabled connection will pass appropriate descriptors to the Source. The Sink will also return the QoS-related capabilities of every protocol, when queried about the supported ones. This information will be used when deciding on which protocol to use. If a successful match is found, the process will go on as usual. Otherwise, the Source will report the failure to the client.

![Connection teardown sequence diagram](image)

**Figure 4-11 - Connection teardown sequence diagram**

### 4.3.4.3 - Multicast

The need to send the same data stream to several destinations is very common in a television environment, such as when a stream is being recorded and monitored at the same time. This requires the ability to connect several Sinks to the same Source.

The model proposed previously already supports this use case, simply by the client requesting connection set-ups several times. However, the actual sending of the data can be performed in two ways: either several independent connections are established, or one multicast connection is created and data are sent to all Sinks simultaneously. Actually, a mix of these two approaches could be used, when some Sinks do not support
multicasting, but the use of multicast should be preferred in order to minimize network usage.

The use of multicast raises a few issues depending on the type of networks and protocols involved. Some technologies, like ATM, support the upgrading of a unicast connection to multicast, while others, such as IP, do not.

Designating all connections as multicast would maximize flexibility, however, for some networks, this would be a very heavyweight process. The conclusion is that policies should be put in place in order to decide the most appropriate approach for each situation. Whenever possible, clients should ‘hint’ the Source when it is expected that several Sinks will be used.

### 4.3.5 - Metadata plane

Metadata management systems are very complex undertakings and, typically, very specific to the institution they are being run on. Thus, it would not be wise to try and define a fully-fledged model for metadata management at this early stage.

![Metadata Plane protocol stack](image)

Figure 4-12 - Metadata Plane protocol stack

Therefore, it was decided to include in the model only the necessary mechanisms for navigation between the metadata and the essence worlds.

Metadata management systems use (or at least will in the near future) UMIDs to identify pieces of essence, whereas in DIMICC object references are used for that purpose.

To perform the link between the two namespaces, there will be a system service responsible for the mapping between a UMID and a CORBA reference, the **Essence Locator Service (ELS)**.
Whenever essence is ingested into the system, or is created within it, it will register with the ELS. If, for any reason, essence moves from one server to another, the entry in the repository will be updated in order to reflect the new object reference.

Clients, upon getting a UMID from the metadata management system, will query the ELS and retrieve the object reference of the desired piece of essence.

Navigability on the other direction will be provided by the ability of querying an essence item for its UMID. In order to do that, a class MediaObject will be defined that has a UMID attribute (see Figure 4-13).

![MediaObject](image)

**Figure 4-13 - The MediaObject interface**

*MediaObject* shall be used as a base class for every one class that represents a media item.
Chapter 5 - DIMICC Implementation

The implementation of DIMICC was divided in two major tasks. The first one was the development of the DIMICC Essence Transfer Already Implemented Library (DETAIL), responsible for the transparent interconnections of Sources and Sinks over different networks. The second task, itself subdivided in smaller ones, was the implementation of the different components, services and clients that would compose the whole system.

The author was responsible for the design and implementation of DETAIL and also collaborated on the design of other parts of the system.

The feedback received from the other developers, all of them of course users of the DIMICC object model and of DETAIL, was extremely useful and, together with the insight gained from the implementation and testing, led to several improvements in both the object model and on DETAIL.

This chapter starts by describing the tools used for the development, followed by a description of the analysis and implementation processes.
5.1 - Tools Used

5.1.1 - ACE

One of the requirements for the implementation of DIMICC was that it should be able to run on Windows and Linux machines and that code should be portable as easily as possible to other platforms.

Of course, everyone who has ever written portable communications software knows that it is a disheartening mission. The differences between system APIs, sometimes just semantic nuances, sometimes completely different interfaces, make it extremely difficult to produce effective and reliable code.

![ACE class hierarchy diagram](source: [41])

It was then decided to look for a framework that shielded the code from these differences and the result was the adoption of the ADAPTIVE Communication Environment (ACE) [7][41].

ACE is a freely available, open-source object-oriented framework that implements many core design patterns for concurrent communication software and provides a set of reusable
C++ wrapper façades and framework components that perform common communication software tasks across a range of OS platforms.

The Distributed Object Computing group at Washington University, in St. Louis, Missouri, is leading ACE development, but several hundred users actively contribute to the code base, in what is one of the best examples of Open Source software development.

ACE mechanisms include event demultiplexing and event handler dispatching, signal handling, service initialisation, interprocess communication, shared memory management, message routing, dynamic (re) configuration of distributed services, concurrent execution and synchronization. Figure 5-1 shows ACE's class hierarchy.

The use of ACE has proved to be fundamental to the success of the implementation by simplifying the programming effort, allowing the use of more complex patterns and augmenting the reliability of the whole system.

Besides, the ACE mailing list is probably one of the best discussion fora on the Internet, giving great insight over a wide range of problems related with communications software programming.

5.1.2 - TAO

The ACE ORB, or TAO, is a CORBA 2.3 compliant ORB built upon ACE.

TAO started as a research project at Washington University and its main goals were to provide a high quality, freely available, open-source ORB. Additionally, there was a great deal of interest in empirically determine the features needed to allow the real-time CORBA ORBs support mission-critical applications with deterministic and statistical QoS requirements. This work led to capturing and documenting the key design patterns and optimisation principle patterns necessary to develop standards-compliant, portable, and extensible QoS-enabled ORBs.

The group that designed TAO has been deeply involved with the OMG work on specifying real-time ORB requirements and specifications, which eventually led to the release of the Real-Time CORBA standard.

TAO comes with several CORBA COS Services, among which are the Telecom Logging Service, the Naming Service, the Notification Service, the Property Service, the Security Service, the Time Service and the Trading Service.
TAO’s block diagram is shown in Figure 5-2 where the real-time-related subsystems are apparent.

Our interest in TAO stem from both its close relation with ACE and its real-time capabilities, which are very important in QoS stringent systems like the ones we find in television.

Unfortunately, when our work began, TAO was still in its infancy and it was decided to use ORBacus (see below). Our experience with ORBacus has been more than satisfactory but work is underway to port part of the implementation to TAO, the rationale being the taking advantage of TAO’s real-time capabilities and its being completely free. Moreover, using two different ORBs serves as a good demonstration of the seamless CORBA interoperability.

5.1.3 - ORBacus

When the implementation started, the need to find a CORBA implementation suitable for our application obviously arose.
After a relatively thorough research of the market, it was decided to use ORBacus [49], a CORBA compliant ORB, produced by Object Oriented Concepts, Inc. [40].

ORBacus is available in both C++ and Java versions, the former being focused on excelling at high load and performance applications, while the latter emphasizes portability and full integration with the Java 2 Platform. Both versions are completely interoperable with each other and with other manufacturers’ products that use IIOP.

The availability of a C++ version was fundamental for the high performance we anticipated the servers would require, whereas the Java version was extremely interesting for the development of less demanding services and graphical user interfaces (GUIs).

The version available at the time the implementation started, ORBacus 3, was a Basic Object Adapter (BOA) based product. Fortunately, shortly after the OMG published the Portable Object Adapter (POA) specification, OOC released ORBacus 4, which is completely compatible with that specification.

An important feature of ORBacus is the excellent support for multithreading, allowing the selection of concurrency models on an object basis.

Additional important features are the support for portable interceptors and dynamic any; the availability of interface and implementation repositories; and the pluggable protocol framework that allows the use of various network protocols.

ORBacus also comes with implementations of the standard Naming, Trading, Notification, Security and Property services. Obviously, this was also very important since, thanks to the way DIMICC is designed, they could be immediately reused.

Finally, a very important factor that also contributed for choosing ORBacus is its being open-source and free for non-commercial use. In addition, there is very responsive support and an excellent mailing list that, together with the ability to read the code, provide excellent learning.
5.2 - DETAIL

5.2.1 - Overview

The DIMICC Essence Transfer Already Implemented Library (DETAIL) is the layer that shields users from dealing with DIMICC's connection management protocol and with the network, providing general implementations of Sources and Sinks that can be used when implementing specialised VirtualComponents.

Although users are free to write their own implementations, factoring out common code in a library leads to better productivity and more reliable systems.

DETAIL has to interface with three actors: the other objects in DIMICC Essence Transfer Plane, the underlying networks (via the OS) and the user who is writing the VirtualComponent (see Figure 4-9 - Essence Transfer Plane protocol stack). These interface points are defined, respectively, by the DIMICC Connection Management Protocol, the OS API and the DETAIL API.

The DETAIL API defines an interface between programmers and DETAIL that is common across different OSs, and is thus fundamental for code portability.

The next sections describe the design and implementation of DETAIL.

5.2.2 - Requirements

1. The main requirements for DETAIL include functional requirements such as:
2. The ability to instantiate Source and Sink objects;
3. The existence of methods for sending and receiving data through these objects;
4. TCP/IP and ATM AAL5 support plus non-disruptive addition of new protocols;
5. Multicast support;
6. The automatic discovery of available networks and protocols but also the provision of means for manually selecting the usable protocols.

And non-functional requirements:

7. Strictly adhere to the DIMICC Connection Management Protocol;
8. Use C++ as the programming language;
9. Be implemented in Linux and Windows and be easily portable to other OSs;

10. Provide the same API across different OSs.

11. Minimize overhead and maximize data throughput;

12. Maximize performance of data transfers between Endpoints in the same process or in the same host;

13. Minimize impact caused on users by changes in the implementation, namely by not requiring recompilation and, if possible, relinking.

5.2.3 - Analysis and Design

Requirement 2 states that users must be able to instantiate Source and Sink objects. This apparently simple requirement implies some thought on the best approach to do it.

![Figure 5-3 - EndpointFactory class diagram](image)

The usual means to instantiate objects in C++ is using constructors [3]. However, the use of constructors requires access to the header file that describes the class. The need to include the header files with the definitions of Source and Sink implementations would force users to recompile their code whenever a minimal change was made to their header [4].

A solution to this problem is possible by applying the **Abstract Factory** pattern [2].
In this particular case, we could define a class `EndpointFactory` with methods for creating `Source` and `Sink` implementations. `AbstractSource` and `AbstractSink` would be abstract classes representing the interface offered by the implementations.

The relationships between these classes are shown in Figure 5-3.

Applying this pattern, implementations are free to change and, as long as they implement the abstract interfaces, clients need not to be aware of such changes.

Regrettably, even if this solution avoids recompiling, the changes in the library will force a relinking. We can go one step further and solve this problem by separating the implementation of the endpoints from the main library and placing them in a dynamically linked library. With this approach, the implementation will be loaded at run time and the user will only have to install a new library whenever a change is made to the code.

This strategy can also be used to satisfy requirement 4, that is, to define an abstract interface for a protocol and use an abstract factory to instantiate dynamically loaded implementations.

![Diagram](image)

**Figure 5-4 - DETAIL component diagram**

In Figure 5-4, it is possible to see the presence of the Endpoint Factory and Transport Factory components, but users will not use those components directly. In order to isolate the user from the complexity of DETAIL, there is a class named `Manager` that acts as a Façade [2] for the different sub-systems.
When DETAIL is initialised, Manager is responsible for starting up the transport and endpoint factories.

The TransportFactory then uses the Property Service in order to find out if there are any configuration parameters related with transport protocols. If not, it tries to find out which ones are available and uses the ACE Service Configurator to load the DLLs that implement the interfaces with each of the transport protocols, the same happening with EndpointFactory that loads the library that contains the endpoint implementations.

The use of ACE Service Configurator permits the complete independence with regard to the underlying OS, maximizing the portability in an area as troublesome as the use of DLLs.

5.2.4 - Performance Optimisation

Several strategies and tactics were used to maximize the performance of DETAIL. Special care was taken with memory copying and concurrency issues.

5.2.4.1 - Memory Copying

As is well noticed in Figure 5-4, there is a sub-system called Message Block Factory which is very important for satisfying requirements 11 and 12.

In systems such as DETAIL, where large amounts of data are passed between different sub-systems, the overhead caused by data copying can be enormous. In order to minimize this, ACE Message Blocks are passed between sub-systems, rather than the data themselves.

ACE Message Blocks are reference-counted pointers to ACE Data Blocks and, while the latter actually contain the data, the former only point to it. Since they are reference counted, they destroy themselves whenever no one is using them anymore and also take care of the destruction of the ACE Data Blocks they point to.

ACE Message and Data Blocks can exist in different flavours, supporting the allocation of memory from the process heap or from system-wide shared memory. This ability can be used to pass blocks of data directly between endpoints located in the same process or in the same machine, without requiring the use of any network transport protocol.
5.2.4.2 - Concurrency

The use of multithreading is fundamental in complex communications applications; unfortunately, if not enough attention is paid, the performance of the whole system can degrade considerably due to gratuitous context switching.

In DETAIL, the transport sub-systems use the ACE Reactor [9] for multiplexing data from different connections, rather than using listening threads on each active connection.

![Diagram showing relationships between classes](image)

**Figure 5-5 – The use of ACE Reactor**

Figure 5-5 shows the relationships between the classes involved in this pattern.

For every active inbound connection, there is an object that implements the class `AbstractInboundConnectionHandler`. Several classes of this kind will exist, one for each of the available protocols.

This object registers itself on the Reactor, which will notify the object whenever new data is available on the connection. The connection handler will process the data and pass it, using an ACE Message Block, to the appropriate Sink.

A similar situation occurs in the outbound path. One obvious difference is that, in this case, the Reactor will notify the connection handler whenever it is possible to send data.

A slightly less obvious difference is the multiplicity of the relationships. In DIMICC, a Sink can only receive data from one Source. Hence, there is a one-to-one relationship between `SinkImplementation` and `InboundConnectionHandler`. On the other hand, a Source can send data to several Sinks, justifying the many-to-one relationship between `SourceImplementation` and `OutboundConnectionHandler`.
5.2.5 - Testing

Testing is a fundamental operation on a software development process. In order to test DETAIL thoroughly, a testing process was set up and used throughout the implementation and deployment phases.

Although more radical approaches exist that stipulate the writing of tests even before the actual implementation begins [43], the method used in DETAIL involved the writing of a test harness per component as soon as an initial implementation was available.

The testing process is hierarchical, in the sense that independent components are tested alone and, only after they pass all the tests, components that use them are tested [4].

![Component dependency graph for hierarchical testing](image)

Figure 5-6 - Component dependency graph for hierarchical testing

Figure 5-6 shows a subset of the DETAIL component diagram where the dependencies between components are apparent. The testing process for these classes involves the stand-alone testing of the Transport Protocol Implementation component, followed by a test in conjunction with Transport Factory. Finally, Endpoint Implementation would be tested in conjunction with the other two modules.

The availability of a structured testing process helped avoiding the typical introduction of new bugs along with the correction of previous ones, or with the addition of new features. Whenever this happened, a new test was written, precluding the same error from appearing in a subsequent iteration.
5.3 - Component and Service Specialisations

Several Component and Service specialisations and client applications were designed and implemented using the DIMICC object model and DETAIL, the most important possibly being the pair Stream Server and Stream Player.

The Stream Server, whose class diagram is shown in Figure 5-7, provides client access to and storage of essence streams in a hard disk.

![Class Diagram](image)

Figure 5-7 - Stream Server class diagram

On the other hand, Stream Player is a client that accesses the server and lets the user browse and play the available streams. Users are also allowed to control the operation of the server, by pressing the usual control buttons, such as play, pause, fast forward/reverse, skip frame, etc.
Obviously, both Stream Server and Stream Player use DETAIL to manage the data connections.

Other components and services were implemented, such as a File Server, an Encoder, a VTR and the Essence Locator Service.

Client applications include a Stream Acquisition GUI, a Stream Server Management GUI and a Stream Monitoring application.
Chapter 6 - Conclusion

Hopefully, this dissertation has succeeded on giving a reasonably thorough overview of the work that led to the current state of DIMICC and its relatives, preceded by a synopsis of the state-of-the-art of television control systems.

An undertaking such as the development of DIMICC has its high and low points and, with any luck, will have its deal of impact on its surroundings.

This chapter points out the lessons learned and the major difficulties encountered during this work, both in a micro-level view, concerned to the project on its own, and on a macro-level view, where its relationship with the whole industry is discussed.

Finally, the author tries to envision the rest of the story. Though it is not an easy task in this complex, yet exhilarating, environment such as the television world...
6.1 - Micro-level view

The development of DIMICC started from a very specific set of requirements, drawn from the experience gained with not very successful previous approaches to the problem in hand. However, the evolution towards a specification that could also be generalized to other environments was much harder a task.

Being an area – the application of distributed systems techniques to television systems control – where absolutely no previous work was published, the requirements elicitation process depended totally on the knowledge of the environment. Which was absolutely none!

Fortunately, the close collaboration with such a reputable and technically competent institution as the BBC provided us with an accurate view of the inner recesses of a television studio environment.

On a personal basis, this opportunity to become familiar with the workings of television was one of the most interesting parts of this work, along with the object-oriented analysis and design that was necessary during the process.

Actually, this was one of the most difficult tasks: the design of an object model that fitted properly the requirements and that was extensible and flexible enough to be adaptable to different environments. Nonetheless, it was also extremely rewarding and a lot of learning was extracted, namely on the correct use of object patterns, an exciting area where very interesting work is currently being done.

Concerning implementation, difficulties crept in from everywhere.

The implementation of DETAIL required the use of complex and highly concurrent communication mechanisms. In this kind of systems, the management of the whole process and the keeping of consistency impose a great deal of concentration on a proper and clean design. When high efficiency is at premium, as is the case, keeping the overhead as low as possible becomes a recipe for headaches.

Fortunately, the use of an excellent tool such as ACE and the application of good design principles and object patterns helped to end up with a decent implementation.

Of course, reliability is sometimes a problem and one must find out how to excuse oneself when bugs start to appear. Or how to correct them...
6.2 - Macro-level view

It has already been stressed out how complex the television world is. There is a lot of highly evolved hardware pieces, excellently designed and with astonishing capabilities. But there is also an enormous amount of complex systems in use, some of them completely outdated, some of them newer, but almost always with strange peculiarities, incompatibilities or just plain bad design.

But people get used to it and, when they get used to it, it is difficult to convince them that there are better ways to do the job.

Probably this is one of the reasons, along with the inertia caused by the huge investments, why IT-based systems are not spreading on TV facilities as they do almost anywhere else. Sure enough, there are lots of computers. But not performing the most demanding tasks, the ones on which the income absolutely depends.

There is still a great deal of preconceptions with regard to computer’s reliability and fitness for real-time stringent tasks such as play-out or on-line editing systems. Some of them may absolutely be justified, some are just remains of the past.

This has been one of the main obstacles found during the projects that were presented and probably one of the most difficult to get over it.

Other problems are related to the size of the typical broadcasting companies. This raises issues such as scalability, for which simulations do not always give reliable results and proper demonstrations are almost impossible.

Size also impacts on the amount of bureaucracy, and on lobbying, and on the preponderance of the “establishment”. But this is a technical document, no politics, please...
6.3 - The Future

Everybody knows it is difficult to predict the evolution of computer technology in general, let alone its impact on an alien ground such as the television control systems.

However, we have been witnessing a growing interest on the application of distributed systems solutions to this area. Oddly, or maybe not, this interest is much more visible on research institutions and on end users - the broadcasters – rather then on the industrial companies themselves. This may suggest a complete lack of interest on this area or, perhaps more likely, their trying to hide their game.

At any rate, there is work underway on standards bodies in order to pave the way for the definition of standards in this area. The most notorious effort is probably the SMPTE ASCA group, which is evolving slowly but steadily. DIMICC has already been presented as input for this group, and some of the ideas will likely be incorporated on its output.

Similarly, DIMICC is also playing an important role on Pro-MPEG’s Networking & Control group. This group is promoting demonstrations of interoperability between devices from different manufacturers using an object-oriented middleware as the glue between these systems. This middleware is based both on DS-CC and on DIMICC and it is expected to provide a good demonstration of the potential of this kind of approach.

Hopefully, demonstrations like this one and the perseverance of end users, at least of the more clear sighted, will lead to the appearance of the first standards and open products, and to the dawn of a new era in the world of television systems control...
Books


Papers


Standards


**Web Pages**


[34] ATSC Web site, http://www.atsc.org


[38] Pro-MPEG Web site, http://www.pro-mpeg.org


    http://www.execpc.com/~gopalan/misc/compare.html

**Miscellaneous**

[46] EBU/SMPTE, *Task Force for Harmonized Standards for the Exchange of

[47] Dare, P., *Migration to the Distributed Object Model-Control*, Sony Corporation
    letter to SMPTE S22.02 Group, 1999.

[48] HDTV Broadcast Collaboration under NIST ATP, *Digital Studio Command and
    Control (DS-CC™) version 2.1*, 1998.

Index

A
AAL.......................... xv, 31, 33
ACE..... xv, xvi, 61, 62, 63, 68, 69, 74, 80
AES.................................. xv, 5, 79
AES-3................................ 5, 79
ATLANTIC...iii, iv, v, vii, 31, 32, 34, 36, 43, 44, 78
ATM......... xv, 26, 27, 31, 58, 65, 89
ATSC.................................. xv, 2, 79

B
BOA ................................ xv, 20, 64

C
CNN......................... 3
COM............................ xvi, 14
CORBA...iii, iv, v, xvi, 13, 14, 15, 16, 17, 18, 20, 21, 26, 29, 36, 41, 42, 44, 45, 47, 48, 49, 53, 54, 55, 58, 62, 63, 64, 79, 87

D
DCOM........ xvi, 13, 14, 26, 79, 80
DIMICC ..viii, xvi, 33, 43, 45, 46, 47, 49, 54, 55, 56, 58, 60, 61, 64, 65, 69, 71, 73, 74, 76, 85, 87, 88, 89, 90, 91, 92, 94, 95, 96, 97, 98, 99, 101
DLL................................. 67, 68
DS-CC... 36, 37, 38, 39, 40, 41, 42, 76, 80
DSM-CC... xv, 27, 28, 29, 33, 36, 42, 44, 45, 78

DVB .................................. xv, 2, 28, 79

E
EBU.......................... xv, 10, 11, 23, 79, 80
 editing ....6, 7, 8, 9, 11, 23, 31, 32, 35, 75
 non-linear................... 8, 40
 on-line............................ 8
EDL.......................... xv, 6, 8, 32, 35
Ethernet.......................... 5, 89

F
FTP.......................... xvi, 33

G
GIOP.......................... xvi, 17

I
IIOP.......................... xvi, 14, 17, 64
 interface
 user................................ xvi, 32, 35, 64, 72
 IOR................................ xvi, 17

J
Java............ xvi, 13, 14, 16, 21, 64, 79

M
metadata iii, iv, 5, 7, 9, 11, 22, 23, 34, 35, 40, 45, 47, 58, 59, 88
MPEG vi, xvi, 2, 8, 11, 26, 27, 28, 31, 32, 76, 80
multicast..................... 45, 55, 57, 58, 65
<table>
<thead>
<tr>
<th>Page</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>NFS</td>
</tr>
<tr>
<td>O</td>
<td>object</td>
</tr>
<tr>
<td></td>
<td>reference</td>
</tr>
<tr>
<td></td>
<td>operating system</td>
</tr>
<tr>
<td></td>
<td>ORB</td>
</tr>
<tr>
<td></td>
<td>ORBacus</td>
</tr>
<tr>
<td></td>
<td>ORBIT</td>
</tr>
<tr>
<td>P</td>
<td>POA</td>
</tr>
<tr>
<td>Q</td>
<td>QoS</td>
</tr>
<tr>
<td>R</td>
<td>RS-422</td>
</tr>
<tr>
<td>S</td>
<td>SDI</td>
</tr>
<tr>
<td></td>
<td>Sky News</td>
</tr>
<tr>
<td>T</td>
<td>TAO</td>
</tr>
<tr>
<td></td>
<td>TCP/IP</td>
</tr>
<tr>
<td>V</td>
<td>VHS</td>
</tr>
<tr>
<td></td>
<td>video on demand</td>
</tr>
<tr>
<td></td>
<td>VIDION</td>
</tr>
<tr>
<td></td>
<td>VTR</td>
</tr>
</tbody>
</table>
Appendix A - Object Model Notation

The object models in this dissertation were drawn using the Unified Modelling Language (UML) [25].

These models use classes called ConcreteXXX to stand for some concrete implementation of interface XXX. These classes are employed to denote semantic relationships between different objects, the reason for this option being that it is not correct to say that an interface uses or creates another interface, but rather that these relationships exist between objects that implement such interfaces.

For example, consider Figure A-1 where class ConcreteVTR implements interface VTR and ConcreteVirtualVTR implements VirtualVTR.

![Figure A-1 - UML diagram](image-url)
The fact that objects that implement interface `VTR` create objects that implement interface `VirtualVTR` is represented by a semantic relationship between `ConcreteVTR` and `ConcreteVirtualVTR`, rather that between the interfaces themselves.
Types.idl

#include "CosProperty.idl"

module CosNotification {
    interface NotificationChannel;
};

module DIMICC {
    interface Source;
    interface Sink;
    interface VirtualComponent;
    interface Device;

    /* This module contains all definitions related to DIMICC components. */

    /* System wide unique identifier. */
    typedef unsigned long DeviceId;

    /* System wide unique identifier. */
    struct NetworkPortId {
        /* Id of the device to which this port is attached. */
        DeviceId theDeviceId;
        /* Device-wide unique identifier. */
        unsigned short theNetworkPortId;
    };

    /* System wide unique identifier. */
    struct ComponentId {
        /* Id of the device where the component is running. */
        DeviceId theDeviceId;
        /* Device-wide unique identifier. */
    };

unsigned short theComponentId;

/* System wide unique identifier. */

struct VirtualComponentId {
    /* Id of the component to which this VC belongs. */
    ComponentId theComponentId;
    /* Component-wide unique identifier. */
    unsigned short theVirtualComponentId;
};

/* System wide unique identifier. */

struct SinkId {
    /* Id of the VC to which this source belongs. */
    VirtualComponentId theVirtualComponentId;
    /* VirtualComponent-wide unique identifier. */
    unsigned short theSinkId;
};

/* System wide unique identifier. */

typedef unsigned long ServiceId;

enum StudioObjectKind {
    SERVICE_KIND,
    COMPONENT_KIND,
    VIRTUAL_COMPONENT_KIND,
    NETWORK_PORT_KIND,
    SOURCE_KIND,
    DEVICE_KIND,
    SINK_KIND
};

/* Carries an object id, independently of the object kind. */

union StudioObjectId switch(StudioObjectKind) {
    case SERVICE_KIND: ServiceId theService;
    case COMPONENT_KIND: ComponentId theComponent;
    case SOURCE_KIND: SourceId theSource;
    case DEVICE_KIND: DeviceId theDevice;
    case VIRTUAL_COMPONENT_KIND: VirtualComponentId
        theVirtualComponent;
    case NETWORK_PORT_KIND: NetworkPortId theNetworkPort;
    case SINK_KIND: SinkId sink;
};

/* Base class for all objects present in the studio. */

interface StudioObject : CosPropertyService::PropertySet {
    /* System-wide object identifier. */
readonly attribute StudioObjectld theStudioObjectld;
    /* Human readable name, such as, "Encoder 1" or "MPEG1 Video Source". */

readonly attribute wstring theStudioObjectName;
    /* Returns the notification channel associated with this object and its owner.
Note that the channel may be shared by several objects. */
    CosNotification::NotificationChannel getNotificationChannel{
        out DIMICC::StudioObjectld owner
    };

    /* Since CORBA IDL does not support exception inheritance, the use of
an exception type per kind of error results in long catch clauses on the client programs. This eventually results in client
writers not catching all the exceptions. DIMICC defines a this global exception type to be used in all the
methods, a solution that is less elegant but more effective. */

exception GeneralException {
    /* Demultiplexes different error types. The valid codes are
defined elsewhere. */
    unsigned long code;
    /* A container for error-specific data. */
    any value;
    /* Human readable description. */
    string description;
};

    /* A list of strings. */
typedef sequence <string> StringList;

    /* A list of anys. */
typedef sequence <any> AnyList;

    /* A pair 'identifier' and 'value'. */
struct NVPair {
    string id;
    any value;
};

    /* A list of NVPairs */
typedef sequence <DIMICC::NVPair> NVPairList;

    /* Encapsulates the address of a network endpoint.
This allows network addresses to be passed regardless of the
protocol. */
struct SAP {
    /* The id of the protocol. The valid ids are defined
elsewhere. */
    unsigned long protocolld;
    /* The protocol-specific address of the endpoint. */
The exact data type is defined in each protocol profile. */
any address;
}

/* A list of network endpoints addresses. */
typedef sequence <SAP> SAPList;

/* Used in FlowSpec to describe a VBR flow. */
struct VBRFlowSpec {
    /* The average bit-rate in bit/s. */
    unsigned long avgBitRate;
    /* The peak bit-rate in bit/s. */
    unsigned long peakBitRate;
}

/* Used in FlowSpec as a switch for different flow types. */
enum FlowType {
    UBR_flow,       /* Bit-rate is unknown. */
    CBR_flow,       /* Bit-rate is constant. */
    VBR_flow       /* Bit-rate is variable, but can be
                    statistically characterized. */
};

/* Describes a data flow, in terms of bit-rate. */
union FlowSpec switch(FlowType) {
    case UBR_flow: unsigned short UBR;    /* Bit-rate is unknown. */
    case CBR_flow: unsigned long CBR;     /* Bit-rate is constant. */
    case VBR_flow: VBRFlowSpec VBR;      /* Bit-rate is variable,
                                           but can be statistically
                                           characterized.
}

/* A service is something that performs some work on behalf of
clients but does not get involved in data flows.
Examples are metadata services, location services, etc. */

interface Service : StudioObject {
};

/* The identifier of a connection. */
struct ConnectionDescriptor {
    /* Id of the source. */
    DIMICC::SourceId theSourceId;
    /* The source of this connection. */
    Source theSource;
    /* The sink of this connection. */
    Sink theSink;
    /* A source-unique identifier. */
    unsigned short localId;
    /* The flow descriptor. */
    FlowSpec theFlowSpec;
    /* The QoS parameters. */
    NVPairList theQoS;
    /* The id of the protocol. The valid ids are defined
elsewhere. */
    unsigned long theProtocolId;
/* The protocol-specific network connection descriptor. */
theNetworkDescriptor;
);
/* A list of connection identifiers. */
typedef sequence <ConnectionDescriptor> ConnectionDescriptorList;
/* A list of sources. */
typedef sequence <Source> SourceList;
/* A list of sinks. */
typedef sequence <Sink> SinkList;
/* A list of VirtualComponents. */
typedef sequence <VirtualComponent> VirtualComponentList;
/* A virtual component is used to control a device and contains the communication endpoints. When a controller wants to control a Component, it requests it a new VirtualComponent. The VirtualComponent retains the state of the current session so that if several clients want to access the same Component, they will not interfere. */
interface VirtualComponent : StudioObject {
    attribute Component theComponent;
    /* Returns the component which created this object. */
    DIMICC::Component getComponent ();
    /* Returns all sources owned by this object. */
    DIMICC::SourceList getSources ();
    /* Returns all sinks owned by this object. */
    DIMICC::SinkList getSinks ();
};
/* A NetworkPort represents a physical network interface, like a Ethernet or ATM card, or a SDI port. */
interface NetworkPort : StudioObject {
    attribute Device theDevice;
    /* Returns the device to which this port is attached. */
    DIMICC::Device getDevice ();
};
/* A list of network ports. */
typedef sequence <NetworkPort> NetworkPortList;
/* A device represents a physical equipment, like a computer, a VTR, or a stand-alone encoder. */

interface Device : StudioObject {
    typedef sequence <NetworkPort> NetworkPorts;

    attribute NetworkPorts theNetworkPorts;

    /* Returns the network ports attached to this device. */
    DIMICC::NetworkPortList getNetworkPorts ();
};

interface Endpoint : StudioObject {
    /* Returns the virtual component that owns this object. */
    DIMICC::VirtualComponent getVirtualComponent ();

    /* Returns the descriptors of all active connections. */
    DIMICC::ConnectionDescriptorList getConnectionDescriptors ();

    /* Returns the network ports attached to the device where this endpoint is running. */
    DIMICC::NetworkPortList getNetworkPorts ();
};

/* A Sink is a communication endpoint that receives data sent to a VirtualComponent. Sinks can be connected to Sources in a point-to-point or point-to-multipoint fashion. */

interface Sink : Endpoint {
    /* Returns the SAP where this sink is accepting connections. */
    DIMICC::SAPList getSAPList (in DIMICC::FlowSpec fs,
                                in DIMICC::NVPairList qosList)
       raises (DIMICC::GeneralException);

    /* Invoked by a source to notify this sink that it is now connected. */
    void connected (in DIMICC::ConnectionDescriptor cd,
                    in DIMICC::FlowSpec fs,
                    in DIMICC::NVPairList qosList)
       raises (DIMICC::GeneralException);

    /* Invoked by a source to notify this sink that it is now disconnected. */
    void disconnected ()
       raises (DIMICC::GeneralException);
};
interface Source : Endpoint {
  /* Invoked by a controller to connect a sink to this source.
     Returns a ConnectionId that is unique */
  DIMICC::ConnectionDescriptor addSink (  
    in DIMICC::Sink aSink,  
    in DIMICC::NVPairList qosList  
  )  
  raises (GeneralException);

  /* Invoked by a controller to disconnect a sink from this source. */
  void removeSink (  
    in DIMICC::ConnectionDescriptor cd  
  )  
  raises (GeneralException);
};
Encoder.idl

#include "DIMICC\Types.idl"

module DIMICC {

    /* This module contains all definitions related to DIMICC components. */

    module Components {

        module EncoderModule {

            /* The VirtualEncoder is be used to control an encoder. */

            interface VirtualEncoder : VirtualComponent {
                /* Returns the source from which data will be sent. */
                DIMICC::Source getSource () ;

                /* Resets the encoder, causes any change on the parameters to be applied. */
                void reset () ;

                /* Starts or resumes encoding. */
                void start () ;

                /* Stops encoding. */
                void stop () ;

                /* Releases the encoder and destroys the VirtualEncoder object. */
                void close () ;
            };

            /* Represents a video or audio encoder. */

            interface Encoder : DIMICC::Component {
                /* Opens the encoder and returns a VirtualEncoder that will be used to control the encoder. If only one encoder is available, it will be locked until the VirtualEncoder is closed. */
                DIMICC::Components::EncoderModule::VirtualEncoder open () ;
            };

            /* Generic exception for the EncoderModule. */

            exception GeneralException {
                /* Demultiplexes different error types. The valid codes are defined elsewhere. */
            };
        }
    }
}
unsigned long code;
/* A container for error-specific data. */
any value;
/* Human readable description. */
string description;
FileServer.idl

#include "DIMICC\Types.idl"

module DIMICC {

    /* This module contains all definitions related to DIMICC
     components. */

    module Components {

        /* This module contains all definitions related to the
         FileServer component. */

        module FileServerModule {

            interface FileServer : Service {
            
            /* References to objects of this type will typically be
             persistent. */

            interface File : DIMICC::Component {
            
            interface VirtualFile : VirtualComponent {

            /* Generic exception for the FileServerModule. */

            exception GeneralException {
                /* Demultiplexes different error types. The valid
                 codes are defined elsewhere. */
                unsigned long code;
                /* A container for error-specific data. */
                any value;
                /* Human readable description. */
                string description;
            };
            
            };
            
            };

        };

    };

};

}
# MediaServer.idl

`#include "DIMICC\Types.idl"`

module DIMICC {

/* This module contains all definitions related to DIMICC components. */

module Components {

/* This module contains all definitions related to the MediaServer component. */

module MediaServerModule {

/* SMPTE UMID. 
(To-do: update this to reflect standard) */

typedef string UMID;

/* This is */

interface MediaObject : DIMICC::Component {
/* The SMPTE UMID assigned to this stream. 
(To-do: check to see if this is consistent with the standard) */

readonly attribute
    DIMICC::Components::MediaServerModule::UMID
    theUMID;
};

/* A list of MediaObjects. */

typedef sequence <MediaObject> MediaObjectList;

interface MediaServer : Service {
/* Returns all objects stored in this server. 
(To-do: use an iterator) */

    DIMICC::Components::MediaServerModule::
    MediaObjectList getObjects ();
};

};

};
StreamServer.idl

#include "DIMICC\Types.idl"
#include "DIMICC\Components\FileStream.idl"
#include "DIMICC\Components\MediaServer.idl"

module DIMICC {
    /* This module contains all definitions related to DIMICC components. */

    module Components {
        /* This module contains all definitions related to the StreamServer component. */

        module StreamServerModule {
            interface MediaStream;
            interface VirtualMediaStream;

            /* A timecode in 90kHz clock ticks. */
            typedef unsigned long long Clock90k;

            /* Generic exception for the StreamServerModule. */
            exception GeneralException {
                /* Demultiplexes different error types. The valid codes are defined elsewhere. */
                unsigned long code;
                /* A container for error-specific data. */
                any value;
                /* Human readable description. */
                string description;
            };

            /* A list of MediaStreams. */
            typedef sequence <MediaStream> StreamList;

            /* A StreamServer is capable of recording and playing out media streams. */

            interface StreamServer : Service {
                /* Returns all streams stored in this server. (To-do: use an iterator) */
                * /
                DIMICC::Components::StreamServerModule::
                StreamList getStreams ();

                /* Creates a new stream object. */
                * /
                DIMICC::Components::StreamServerModule::
                MediaStream createStream ( in DIMICC::Components::MediaServerModule::
                UMID theUMID,
in DIMICC::NVPairList options
"
raises (DIMICC::
    Components::StreamServerModule::
    GeneralException);
"

/* Records a new stream.
References to objects of this type will typically be transient. */

interface StreamRecorder : VirtualComponent {
    /* Returns the sink where data should be sent to. */
    DIMICC::Sink getSink();

    /* Returns the new MediaStream created by this recorder. */
    DIMICC::Components::StreamServerModule::
        MediaStream getStream();

    /* Starts recording immediately. */
    void startRecordingNow();

    raises (DIMICC::Components::
        StreamServerModule::GeneralException);

    /* Starts recording after 'clock' has been reached. */
    void startRecordingAt (in Clock90k clock
        raises (DIMICC::Components::
            StreamServerModule::GeneralException);

    /* Pauses recording immediately. */
    void pauseRecordingNow();

    raises (DIMICC::Components::
        StreamServerModule::GeneralException);

    /* Pauses recording after 'clock' has been reached. */
    void pauseRecordingAt (in Clock90k clock
        raises (DIMICC::Components::
            StreamServerModule::GeneralException);

    /* Releases the resources associated with this recorder. */
    void close();
/* Aborts the recording and returns the stream to 
EMPTY state. */
void abort();

interface MediaStream : MediaServerModule::MediaObject 
{
    /* The possible states of a MediaStream. */
    enum State {
    RECORDING,
    /* The stream is currently being recorded. */
    MARKED_FOR_DELETION,
    /* Someone asked the stream to be deleted, 
however it is still being used by some user. 
Opens will be disallowed. */
    EMPTY,
    COMPLETE
    /* The stream is completely recorded. */
    
    }
    /* The current state of the stream. */
    readonly attribute State currentState;

    /* Opens the stream as a VirtualMediaStream. */

    DIMICC::Components::StreamServerModule::
    VirtualMediaStream openAsStream ( 
    in DIMICC::NVPairList options 
    ) 
    raises (DIMICC::Components::
    StreamServerModule::GeneralException);

    /* Opens the stream as a VirtualFile. */

    DIMICC::Components::FileStreamModule::
    VirtualFile openAsFile ( 
    in DIMICC::NVPairList options 
    ) 
    raises (DIMICC::Components::
    StreamServerModule::GeneralException);

    /* Opens the stream for recording. Only valid if 
state = EMPTY. State will be RECORDING. */

    DIMICC::Components::StreamServerModule::
    StreamRecorder openForRecording ( 
    in DIMICC::NVPairList options 
    );

    /* Destroys physically the stream. If the stream 
is not in use, it will be immediately deleted. */
Otherwise, state will be MARKED_FOR_DELETION
until all clients finish.
*/
void delete ()
  raises (DIMICC::Components::
    StreamServerModule::GeneralException);
};

/* Created on demand by users to play and control a
stream. References to objects of this type will typically be
 transient. */

interface VirtualMediaStream : VirtualComponent {
  /* The scale at which a stream will be played. A
  positive numerator means playing forward. A
  negative one, backwards. */

  struct Scale {
    short numerator;
    unsigned short denominator;
  };

  /* Encapsulates the current state of the
  VirtualStream. */

  struct State {
    Clock90k current_position;
    Scale current_scale;
  };

  /* The current state. */

  readonly attribute State currentState;

  /* Returns the source from which data will be
  sent. */
  DIMICC::Source getSource ();

  /* Resumes playing at the specified timecode and
  at the specified rate. */
  void resume {
    in Clock90k start_clock,
    in Scale the_scale
  } raises (DIMICC::Components::
    StreamServerModule::GeneralException);

  /* Pauses playing when the specified timecode is
  reached. */
  void pause {
    in Clock90k stop_clock
  };

  /* Pause playing immediately. */
  void pauseNow ();
/* Plays the stream between the specified
timecodes and at the specified rate.
*/
void play {
    in Clock90k start_clock,
    in Clock90k stop_clock,
    in Scale the_scale
};

/* Closes the resources associated with this
VirtualMediaStream. This object no longer can be
accessed after this call.
*/
void close();
VTR.idl

#include "DIMICC\Types.idl"

module DIMICC {

    module Components {

        module VTRModule {
            interface VirtualVTR;

            /* Generic exception for the VTRModule. */

            exception GeneralException {
                /* Demultiplexes different error types.
                The valid codes are defined elsewhere. */
                unsigned long code;
                /* A container for error-specific data. */
                any value;
                /* Human readable description. */
                string description;
            };

            /* Represents a physical VTR and acts as a factory of
            VirtualVTR. */

            interface VTR : DIMICC::Component {
                /* Opens the VTR and returns a VirtualVTR that
                will be used for controlling the VTR. */

                DIMICC::Components::VTRModule::VirtualVTR open ()
                raises (DIMICC::Components::
                        VTRModule::GeneralException);
            };

            /* Used to command the VTR. */

            interface VirtualVTR : VirtualComponent {
                typedef unsigned long Timecode;

                enum VTRState {
                    STOPPED,
                    OPEN,
                    RECORDING,
                    PAUSED,
                    SEEKING,
                    PLAYING,
                    NOT_READY
                };

                /* Plays the tape. */
                void play {
                    in unsigned long speed,
                    in boolean direction_forward
                };

                /* Stops the tape. */

            };

        };

    };

}
*/
void stop ();

/∗ Pauses the tape.
/∗
void pause ();

/∗ Resumes at the previous speed.
/∗
void resume ();

/∗ Moves the tape forward at the maximum speed.
/∗
void fastForward ();

/∗ Moves the tape backward at the maximum speed.
/∗
void fastRewind ();

/∗ Jumps to the specified timecode.
/∗
void gotoTimecode {
    in Timecode position
};

/∗ Steps one frame, forward or backward.
/∗
void step {
    in boolean forward
};

/∗ Ejects the cassette.
/∗
void eject ();

/∗ Returns the current timecode.
/∗
Timecode getCurrentTimecode ();

/∗ Retrieves the current state of the VTR.
/∗
VTRState getCurrentState ();

/∗ Releases the VTR and destroys this object.
/∗
void close ();