Dynamic Configuration of Distributed Multimedia Components

Robert Scott Mitchell
DYNAMIC CONFIGURATION OF DISTRIBUTED MULTIMEDIA COMPONENTS

ROBERT SCOTT MITCHELL

DEPARTMENT OF COMPUTER SCIENCE
QUEEN MARY AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

A THESIS SUBMITTED TO THE UNIVERSITY OF LONDON IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
AUGUST 2000
Author's contact address:
Scott Mitchell
Department of Computer Science
Queen Mary and Westfield College
London E1 4NS
United Kingdom
<scott@des.qmw.ac.uk>

All rights reserved.
Typeset in Bitstream Elegant Garamond by l4X and l5TPX.
All trademarks and registered trademarks mentioned in this work are acknowledged to be the property of their respective owners.
The use of multimedia in distributed computing systems has begun to include such complex and mission-critical application domains as digital television production, "video-on-demand" services, medical and security systems. Continuous-media systems in general impose strong requirements on their underlying operating system and network support mechanisms; this is especially relevant to mission-critical systems where life or property may be endangered if the application cannot maintain a required level of service throughout its lifetime.

Application components are potentially long-lived and may need to operate in a resource-limited environment. Consequently, applications are likely to undergo frequent reconfiguration: to upgrade or enhance existing components; in response to changing environmental conditions (e.g. resource availability); or as a result of user actions. Reconfigurations must be performed while the application is running, with minimal disruption to its ongoing operation. Specifically, the temporal integrity of active continuous-media streams must be maintained across reconfigurations wherever possible.

This thesis presents a novel approach to the construction and dynamic reconfiguration of distributed multimedia applications. The major contributions of this work are:

- **Application modelling.** Applications are composed from hierarchically structured components. A layered architecture separates the low-level concerns of controlling continuous-media stream and devices from application configuration and management tasks. The upper layer of components is a dynamic runtime model of the "active" application's structural, quality-of-service (QoS) characteristics and reconfiguration properties.

- **Atomic actions.** Among the most important requirements for reconfiguration are atomicity and consistency. These properties are maintained by structuring reconfiguration as atomic actions on the application model; the active component layer will not be modified until the reconfigured model has passed structural consistency and QoS admission control tests.

- **Reconfiguration scheduling.** An atomic action results in a set of operations to be applied to the active components of an application. These operations should be performed in such a way as to minimise disruption to active media streams; that is, to maximise the temporal
integrity or "smoothness" of the application's streams during the reconfiguration. I describe
an scheduling algorithm for active component updates that is generic for certain classes of
applications, and explore the tradeoff between reconfiguration time, temporal consistency
and resource usage entailed by this algorithm.

The integration of the application modelling and reconfiguration mechanisms into the DJINN mul-
timedia programming framework is presented. The system is evaluated in the context of two real-
world application scenarios: a security-monitoring system being tested by the emergency services;
and a hypothetical compressed audio/video switching and editing system based on technology de-
veloped in the digital television industry.
The research presented in this thesis was conducted as part of the DJINN project in the Distributed Systems group of the Department of Computer Science, Queen Mary and Westfield College, University of London. The project commenced in October 1995 and involved two first year Ph.D. students, myself (Scott Mitchell) and Hani Naguib, and their supervisors, Prof. George Coulouris and Dr. Tim Kindberg respectively.

One of the major contributions of the DJINN project has been the design and prototype implementation of a programming framework for distributed multimedia systems, the DJINN framework. The initial development of the framework, including the first working prototype and demonstration applications, occupied the following year. During this time, the basic ideas of application modelling that have become a key feature of the DJINN approach were first proposed and investigated. That research was carried out jointly and cooperatively by all members of the DJINN project team and it is thus impossible to assign individual credit to any piece of work. That shared work is discussed in Sections 4.1 and 4.2 (up to 4.2.4) of this thesis. Also, the properties of multimedia applications in Section 3.2 and some of the software requirements in Section 3.3 were developed jointly during the first year of the project. The remainder of the research presented here is entirely my own individual work. Development of DJINN continued with a common code-base for approximately a year after the first prototype, with Mr. Naguib and myself working on separate aspects of the architecture.
ACKNOWLEDGEMENTS

This thesis would not have been possible without the support and advice of a number of people.

Firstly, I must thank my Ph.D. supervisor, Professor George Coulouris, for giving me the opportunity to study at QMW, for his boundless patience and for the encouragement necessary to see this research through to its conclusion.

I am very grateful to the other members of my supervisory panel, Dr. Tim Kindberg and Dr. Sylvia Wilbur, and to my colleagues in the Distributed Systems Laboratory for their advice and many useful discussions. Special thanks are due to Hani Naguib, my collaborator on the DJINN project during my years at QMW. Dr. John Bates, Dr. Mark Spiteri and my other colleagues in the Laboratory for Communications Engineering at the University of Cambridge have also provided invaluable support and motivation.

I acknowledge the numerous helpful comments and suggestions received from my reviewers, and thank them again for their time: Dr. Alan Abrahams, Dr. Martin Brown, Hani Naguib, Dr. Mark Spiteri, Dr. Pete Steggles and James Weatherall.

This research was funded for three years by a Commonwealth Scholarship awarded by the Association for Commonwealth Universities and the New Zealand Government.

Finally, my deepest and most heartfelt thanks must go to my parents, Bob and Julie Mitchell, who have supported me in every way throughout my academic career and always encouraged me to seek new challenges. Likewise, I am forever grateful to my wife, Anne-Lise Heinrichs, for her uncomplaining love and support even during those times when my work has kept us apart.
“Djinn (or Jinn) (from Muslim mythology)
A spirit lower than angels, able to appear in human and animal forms, and having supernatural powers over men.”

Concise Oxford Dictionary
# Table of Contents

1 Introduction 1

1.1 Motivation ........................................ 2
1.1.1 Multimedia applications are real-time applications 3
1.1.2 Multimedia applications are dynamic .............. 4
1.1.3 Distributed multimedia requires high-level support 5

1.2 Contributions ........................................ 6
1.3 Thesis overview ...................................... 7

2 Related Work .......................................... 9

2.1 Multimedia programming frameworks .................. 9
2.1.1 MIT VuSystem .................................. 10
2.1.2 Berkeley CMT .................................. 12
2.1.3 Gibbs and Tischritzis ............................ 15
2.1.4 Medusa and Pandora .............................. 16
2.1.5 Java Media Framework ............................ 17
2.1.6 Microsoft DirectShow ............................ 19
2.1.7 CORBA audio/video streaming specification .. 20
2.1.8 Discussion ...................................... 21

2.2 Distributed system composition and reconfiguration 23
2.2.1 Client-server reconfiguration in Equus .......... 23
2.2.2 Self-adaptive software for DSP applications .... 25
2.2.3 Distributed systems reconfiguration at Imperial College 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.4</td>
<td>The SIRAC project and Olan</td>
<td>32</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Discussion</td>
<td>34</td>
</tr>
<tr>
<td>2.3</td>
<td>Adaptive and reconfigurable multimedia systems</td>
<td>34</td>
</tr>
<tr>
<td>2.3.1</td>
<td>CINEMA</td>
<td>35</td>
</tr>
<tr>
<td>2.3.2</td>
<td>QUASAR</td>
<td>37</td>
</tr>
<tr>
<td>2.3.3</td>
<td>ADAPT</td>
<td>40</td>
</tr>
<tr>
<td>2.3.4</td>
<td>ATLANTIC</td>
<td>42</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Discussion</td>
<td>46</td>
</tr>
<tr>
<td>2.4</td>
<td>Summary</td>
<td>47</td>
</tr>
<tr>
<td>3</td>
<td>REQUIREMENTS STUDIES</td>
<td>49</td>
</tr>
<tr>
<td>3.1</td>
<td>Application Case Studies</td>
<td>50</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Case Study 1: Remote Surveillance</td>
<td>51</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Case Study 2: Digital Television Production</td>
<td>53</td>
</tr>
<tr>
<td>3.2</td>
<td>Properties of distributed multimedia applications</td>
<td>58</td>
</tr>
<tr>
<td>3.3</td>
<td>Software requirements for distributed multimedia</td>
<td>59</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Framework design requirements</td>
<td>60</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Programming model</td>
<td>62</td>
</tr>
<tr>
<td>3.3.3</td>
<td>System support requirements</td>
<td>65</td>
</tr>
<tr>
<td>3.4</td>
<td>Requirements for reconfiguration</td>
<td>66</td>
</tr>
<tr>
<td>3.5</td>
<td>Summary</td>
<td>68</td>
</tr>
<tr>
<td>4</td>
<td>THE DJINN ARCHITECTURE</td>
<td>69</td>
</tr>
<tr>
<td>4.1</td>
<td>Motivation</td>
<td>70</td>
</tr>
<tr>
<td>4.2</td>
<td>Conceptual architecture</td>
<td>71</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Components and ports</td>
<td>72</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Connectors</td>
<td>73</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Application modelling</td>
<td>75</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Application composition</td>
<td>77</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Modelling end-to-end properties</td>
<td>81</td>
</tr>
</tbody>
</table>
5 HIGH-LEVEL RECONFIGURATION ARCHITECTURE

5.1 Defining Reconfiguration

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1</td>
<td>Case study 1: Remote Surveillance System</td>
<td>114</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Case study 2: Digital Television Production</td>
<td>117</td>
</tr>
<tr>
<td>5.1.3</td>
<td>A more precise definition of reconfiguration</td>
<td>120</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Consistent state</td>
<td>121</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Performing the transition</td>
<td>123</td>
</tr>
</tbody>
</table>

5.2 Atomic actions

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.1</td>
<td>Assumptions and design goals</td>
<td>126</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Properties of atomic actions</td>
<td>128</td>
</tr>
</tbody>
</table>

5.3 Atomic action implementation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.1</td>
<td>Package overview</td>
<td>130</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Concurrency control</td>
<td>135</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Consistency validation</td>
<td>136</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Difficulties with Java implementation</td>
<td>136</td>
</tr>
</tbody>
</table>

5.4 Reconfiguring the sample applications

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
</table>

5.5 Summary

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>142</td>
</tr>
</tbody>
</table>
## Table of Contents

6 **RECONFIGURATION SCHEDULING** 143

6.1 Motivation ................................................. 144

6.1.1 High-level view of smoothness .......................... 147

6.1.2 Dynamic consistency as a continuous variable ............ 149

6.2 Algorithms .................................................. 150

6.2.1 Two-stage updates ....................................... 152

6.2.2 Integration scheduling ................................... 161

6.2.3 Extension to multiple paths ............................... 169

6.3 Implementation ............................................. 170

6.3.1 Managing the active layer update ......................... 170

6.3.2 Model→active interface .................................. 171

6.3.3 A complete example ...................................... 172

6.4 Summary .................................................... 176

7 **EVALUATION** 181

7.1 Methodology .................................................. 182

7.1.1 Current framework implementation ....................... 182

7.1.2 Test application design .................................. 183

7.1.3 Experimental design ..................................... 186

7.2 Results and discussion ..................................... 189

7.2.1 Non-optimised trials ..................................... 189

7.2.2 Active object behaviour .................................. 206

7.2.3 Optimised trials .......................................... 208

7.3 Summary ..................................................... 216

8 **SUMMARY AND CONCLUSIONS** 217

8.1 Summary ..................................................... 217

8.1.1 Contributions ............................................. 218

8.1.2 Unresolved issues ........................................ 219

8.2 Directions for future work ................................. 221

8.3 Concluding remarks ........................................ 223
A EXPERIMENTAL RESULTS

A.1 Non-optimised trials ........................................... 225
  A.1.1 Baseline operation ...................................... 225
  A.1.2 Non-structural reconfiguration ......................... 232
  A.1.3 "Change source" reconfiguration ....................... 239
  A.1.4 "Switch encodings" reconfiguration .................... 243
  A.1.5 "Move client” reconfiguration ........................ 248
A.2 Optimised trials .............................................. 255
  A.2.1 "Change source” reconfiguration ........................ 255
  A.2.2 "Switch encodings” reconfiguration .................... 262
  A.2.3 "Move client” reconfiguration ........................ 272
LIST OF FIGURES

2.1 VuSystem design approach (from [LWT94]). ........................................ 10
2.2 JMF media player states (from [JMF98]). ............................................. 18
2.3 SWiFT feedback control model (from [GSPW98]). ................................. 38
2.4 Open binding model (from [FBC+98a]). ............................................... 41
2.5 The ATLANTIC switch (from [KW97]). .................................................. 45

3.1 Overview of the remote surveillance application. .................................... 52
3.2 The ATLANTIC editing facility. .............................................................. 54

4.1 DJINN architectural overview. ............................................................... 71
4.2 A basic DJINN application. ................................................................. 74
4.3 A basic distributed application. ............................................................ 74
4.4 DJINN two-level application structure. ................................................ 76
4.5 A federated application model. ............................................................. 79
4.6 Paths encapsulate end-to-end properties of applications. ...................... 82
4.7 Computing paths from component I/O dependencies. ............................ 83
4.8 Model object state transitions. ............................................................. 86
4.9 Active object state transitions, as seen by the model. ......................... 87
4.10 DJINN top-level package hierarchy. .................................................... 89
4.11 Class hierarchy for package djinn. ...................................................... 90
4.12 Class djinn.DjinnObject. ................................................................. 91
4.13 Class djinn.DjinnRuntime. .............................................................. 91
4.14 Class djinn.DjinnFactory. .............................................................. 92
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.9</td>
<td>Implementation of the <code>powerOn</code> method.</td>
<td>139</td>
</tr>
<tr>
<td>5.10</td>
<td>Switching video sources in the surveillance application.</td>
<td>140</td>
</tr>
<tr>
<td>5.11</td>
<td>Switching networks in the surveillance application.</td>
<td>141</td>
</tr>
<tr>
<td>6.1</td>
<td>A simple active layer reconfiguration.</td>
<td>147</td>
</tr>
<tr>
<td>6.2</td>
<td>Possible reconfigurations of a single path.</td>
<td>149</td>
</tr>
<tr>
<td>6.3</td>
<td>Reconfiguration of adaptive video connector.</td>
<td>151</td>
</tr>
<tr>
<td>6.4</td>
<td>Lifetime of a two-stage update.</td>
<td>154</td>
</tr>
<tr>
<td>6.5</td>
<td>Computing shared and divergent path regions.</td>
<td>156</td>
</tr>
<tr>
<td>6.6</td>
<td>Regions of path overlap and divergence.</td>
<td>157</td>
</tr>
<tr>
<td>6.7</td>
<td>Path regions requiring reconfiguration.</td>
<td>157</td>
</tr>
<tr>
<td>6.8</td>
<td>Computing reconfigured path regions.</td>
<td>158</td>
</tr>
<tr>
<td>6.9</td>
<td>Computing setup/integration stage boundary.</td>
<td>159</td>
</tr>
<tr>
<td>6.10</td>
<td>Setup/integration actions for basic scheduling algorithm.</td>
<td>160</td>
</tr>
<tr>
<td>6.11</td>
<td>Integration regions in the example reconfiguration.</td>
<td>162</td>
</tr>
<tr>
<td>6.12</td>
<td>Basic integration scheduling process.</td>
<td>163</td>
</tr>
<tr>
<td>6.13</td>
<td>Setup/integration actions with adjusted stage boundary.</td>
<td>166</td>
</tr>
<tr>
<td>6.14</td>
<td>Setup/integration actions with optimised integration scheduling.</td>
<td>168</td>
</tr>
<tr>
<td>6.15</td>
<td>Executing integration stage, $t = 0$.</td>
<td>173</td>
</tr>
<tr>
<td>6.16</td>
<td>Executing integration stage, $t = 1$.</td>
<td>174</td>
</tr>
<tr>
<td>6.17</td>
<td>Executing integration stage, $t = 2$.</td>
<td>175</td>
</tr>
<tr>
<td>6.18</td>
<td>Executing integration stage, $t = 4$.</td>
<td>177</td>
</tr>
<tr>
<td>6.19</td>
<td>Executing integration stage, $t = 5$.</td>
<td>178</td>
</tr>
<tr>
<td>6.20</td>
<td>Executing integration stage, $t = 6$.</td>
<td>179</td>
</tr>
<tr>
<td>7.1</td>
<td>Structure of the experimental test application.</td>
<td>184</td>
</tr>
<tr>
<td>7.2</td>
<td>“Change source” reconfiguration on test application.</td>
<td>185</td>
</tr>
<tr>
<td>7.3</td>
<td>“Switch encoding” reconfiguration on test application.</td>
<td>185</td>
</tr>
<tr>
<td>7.4</td>
<td>“Move client” reconfiguration on test application.</td>
<td>186</td>
</tr>
<tr>
<td>7.5</td>
<td>Test application screenshot.</td>
<td>188</td>
</tr>
</tbody>
</table>
List of Figures

7.6 Media rate for baseline operation with MPEG stream. .......................... 190
7.7 End-to-end latency for baseline operation with MPEG stream. .................. 191
7.8 End-to-end jitter for baseline operation with MPEG stream. .................... 192
7.9 Baseline operation with M-JPEG stream. ........................................ 193
7.10 Non-structural reconfiguration with MPEG stream. .............................. 196
7.11 Non-structural reconfiguration with M-JPEG stream. ............................ 197
7.12 "Change source" reconfiguration with MPEG streams. ............................ 199
7.13 "Switch encodings" reconfiguration for M-JPEG→MPEG transition. .............. 202
7.14 "Switch encodings" reconfiguration for MPEG→M-JPEG transition. ............ 203
7.15 "Move client" reconfiguration with MPEG stream. ................................ 204
7.16 "Move client" reconfiguration with M-JPEG stream. ............................... 205
7.17 End-to-end latency for optimised "change source" reconfiguration with MPEG streams. .......................... 209
7.18 End-to-end latency for optimised "switch encodings" reconfiguration for M-JPEG→MPEG transition. .......................... 211
7.19 End-to-end latency for optimised "switch encodings" reconfiguration for MPEG→M-JPEG transition. .......................... 212
7.20 End-to-end latency for optimised "move client" reconfiguration with MPEG stream. .......................... 214
7.21 End-to-end latency for optimised "move client" reconfiguration with M-JPEG stream. .......................... 215
A.1 Media rate for baseline operation with MPEG stream. ............................ 226
A.2 End-to-end latency for baseline operation with MPEG stream. .................. 227
A.3 End-to-end jitter for baseline operation with MPEG stream. .................... 228
A.4 Media rate for baseline operation with M-JPEG stream. .......................... 229
A.5 End-to-end latency for baseline operation with M-JPEG stream. .................. 230
A.6 End-to-end jitter for baseline operation with M-JPEG stream. .................. 231
A.7 Media rate for non-structural reconfiguration with MPEG stream. ............... 233
A.8 End-to-end latency for non-structural reconfiguration with MPEG stream. ........ 234
A.9 End-to-end jitter for non-structural reconfiguration with MPEG stream. ........ 235
A.10 Media rate for non-structural reconfiguration with M-JPEG stream. .......... 236
A.11 End-to-end latency for non-structural reconfiguration with M-JPEG stream. 237
A.12 End-to-end jitter for non-structural reconfiguration with M-JPEG stream. 238
A.13 Media rate for "change source" reconfiguration with MPEG streams. 240
A.14 End-to-end latency for "change source" reconfiguration with MPEG streams. 241
A.15 End-to-end jitter for "change source" reconfiguration with MPEG streams. 242
A.16 Media rate for "switch encodings" reconfiguration, complete trial. 243
A.17 End-to-end latency for "switch encodings" reconfiguration, complete trial. 244
A.18 End-to-end jitter for "switch encodings" reconfiguration, complete trial. 244
A.19 Media rate for "switch encodings" reconfiguration, detail. 245
A.20 End-to-end latency for "switch encodings" reconfiguration, detail. 246
A.21 End-to-end jitter for "switch encodings" reconfiguration, detail. 247
A.22 Media rate for "move client" reconfiguration with MPEG stream. 249
A.23 End-to-end latency for "move client" reconfiguration with MPEG stream. 250
A.24 End-to-end jitter for "move client" reconfiguration with MPEG stream. 251
A.25 Media rate for "move client" reconfiguration with M-JPEG stream. 252
A.26 End-to-end latency for "move client" reconfiguration with M-JPEG stream. 253
A.27 End-to-end jitter for "move client" reconfiguration with M-JPEG stream. 254
A.28 Media rate for "change source" reconfiguration with MPEG streams, scheduling optimisation only. 256
A.29 End-to-end latency for "change source" reconfiguration with MPEG streams, scheduling optimisation only. 257
A.30 End-to-end jitter for "change source" reconfiguration with MPEG streams, scheduling optimisation only. 258
A.31 Media rate for "change source" reconfiguration with MPEG streams, scheduling and early-integrate optimisations. 259
A.32 End-to-end latency for "change source" reconfiguration with MPEG streams, scheduling and early-integrate optimisations. 260
A.33 End-to-end jitter for "change source" reconfiguration with MPEG streams, scheduling and early-integrate optimisations. 261
A.34 Media rate for "switch encodings" reconfiguration with scheduling optimisation, complete trial. 262
<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.35</td>
<td>End-to-end latency for “switch encodings” reconfiguration with scheduling optimisation, complete trial.</td>
<td>263</td>
</tr>
<tr>
<td>A.36</td>
<td>End-to-end jitter for “switch encodings” reconfiguration with scheduling optimisation, complete trial.</td>
<td>263</td>
</tr>
<tr>
<td>A.37</td>
<td>Media rate for “switch encodings” reconfiguration with scheduling optimisation, detail.</td>
<td>264</td>
</tr>
<tr>
<td>A.38</td>
<td>End-to-end latency for “switch encodings” reconfiguration with scheduling optimisation, detail.</td>
<td>265</td>
</tr>
<tr>
<td>A.39</td>
<td>End-to-end jitter for “switch encodings” reconfiguration with scheduling optimisation, detail.</td>
<td>266</td>
</tr>
<tr>
<td>A.40</td>
<td>Media rate for “switch encodings” reconfiguration with scheduling and early-integrate optimisations, complete trial.</td>
<td>267</td>
</tr>
<tr>
<td>A.41</td>
<td>End-to-end latency for “switch encodings” reconfiguration with scheduling and early-integrate optimisations, complete trial.</td>
<td>268</td>
</tr>
<tr>
<td>A.42</td>
<td>End-to-end jitter for “switch encodings” reconfiguration with scheduling and early-integrate optimisations, complete trial.</td>
<td>268</td>
</tr>
<tr>
<td>A.43</td>
<td>Media rate for “switch encodings” reconfiguration with scheduling and early-integrate optimisations, detail.</td>
<td>269</td>
</tr>
<tr>
<td>A.44</td>
<td>End-to-end latency for “switch encodings” reconfiguration with scheduling and early-integrate optimisations, detail.</td>
<td>270</td>
</tr>
<tr>
<td>A.45</td>
<td>End-to-end jitter for “switch encodings” reconfiguration with scheduling and early-integrate optimisations, detail.</td>
<td>271</td>
</tr>
<tr>
<td>A.46</td>
<td>Media rate for “move client” reconfiguration with MPEG stream, scheduling optimisation only.</td>
<td>273</td>
</tr>
<tr>
<td>A.47</td>
<td>End-to-end latency for “move client” reconfiguration with MPEG stream, scheduling reconfiguration only.</td>
<td>274</td>
</tr>
<tr>
<td>A.48</td>
<td>End-to-end jitter for “move client” reconfiguration with MPEG stream, scheduling reconfiguration only.</td>
<td>275</td>
</tr>
<tr>
<td>A.49</td>
<td>Media rate for “move client” reconfiguration with M-JPEG stream, scheduling optimisation only.</td>
<td>276</td>
</tr>
<tr>
<td>A.50</td>
<td>End-to-end latency for “move client” reconfiguration with M-JPEG stream, scheduling optimisation only.</td>
<td>277</td>
</tr>
<tr>
<td>A.51</td>
<td>End-to-end jitter for “move client” reconfiguration with M-JPEG stream, scheduling optimisation only.</td>
<td>278</td>
</tr>
</tbody>
</table>
A.52 Media rate for "move client" reconfiguration with MPEG stream, scheduling and early-integrate optimisations. .................................................. 279
A.53 End-to-end latency for "move client" reconfiguration with MPEG stream, scheduling and early-integrate optimisations. .............................. 280
A.54 End-to-end jitter for "move client" reconfiguration with MPEG stream, scheduling and early-integrate optimisations. .............................. 281
A.55 Media rate for "move client" reconfiguration with M-JPEG stream, scheduling and early-integrate optimisations. .............................. 282
A.56 End-to-end latency for "move client" reconfiguration with M-JPEG stream, scheduling and early-integrate optimisations. ......................... 283
A.57 End-to-end jitter for "move client" reconfiguration with M-JPEG stream, scheduling and early-integrate optimisations. .............................. 284
LIST OF TABLES

3.1 Typical stream parameters for digital television. ......................... 57

5.1 Properties and semantics of atomic actions. .......................... 131

6.1 Path activity states. ................................................. 153

7.1 Latency results for MPEG decoder component. ........................ 207

7.2 Latency results for M-JPEG decoder component. ...................... 207

7.3 Latency results for multicast UDP sink component. ................. 207
CHAPTER 1

INTRODUCTION

Multimedia systems combine artifacts from different media types to enhance communication and enrich the presentation of information [GT95]. Media types include continuous media with an inherent temporal dimension such as digitised audio or video, music and animation, and discrete media such as still images or text that carry no implicit temporal information. However, timing of the presentation of even discrete media elements is generally important to the overall multimedia experience.

The past decade has seen digital multimedia applications evolve from simple, low quality systems running on a single machine, to streaming audio and video across the Internet and the delivery of high quality digital television to millions of viewers. These advances have been driven by the availability of ever-cheaper and more powerful computing hardware and the rise of the Internet as a medium for global communication. The combination of multimedia with communications networks is particularly significant, as it brings these applications within the realm of distributed systems.

More recently, applications of distributed continuous media systems have begun to include more complex, mission-critical domains such as digital television production, commercial "video-on-demand" services, medical applications and security systems. Because of the enrichment they bring to new application areas I believe that this trend will continue and that more and more distributed mission-critical applications will incorporate continuous media data. By their very nature, these applications impose more stringent requirements on the support mechanisms provided by underlying networks and operating systems than currently more widely deployed multimedia applications such as video-conferencing, streaming audio and video and (non-distributed) entertainment and educational software. The quality of the media being presented is important—sometimes critically so—and thus resources must be properly allocated and scheduled in order to preserve this quality at an acceptable level. This thesis is concerned with the configuration and runtime reconfiguration of long-lived, highly dynamic applications operating in domains where maintaining the quality of
service delivered by the application is of paramount importance.

1.1 Motivation

Non-distributed multimedia applications run on a single computer system, usually a personal computer or workstation running a conventional interactive operating system. The multimedia data used by the application are stored or generated by local I/O devices, for example a CD-ROM drive or a directly connected video camera. In general, the primary function of the application will be to manage the presentation—that is, the spatial and temporal arrangement—of the various media data on one or more output devices. The application may be "interactive", although the interaction typically only extends to allowing the single user of the application to modify the presentation in predetermined ways, or to provide additional media data where "live" input devices are available. Moreover, media data processing is often expensive in terms of system resources; multimedia applications often assume that they have exclusive access to the resources of the underlying computer system, and will fail to operate correctly if this assumption is invalid.

In the distributed case, similar considerations apply to the application as a whole with respect to media presentation and user interaction. However, several important additional properties emerge when multimedia applications are distributed. Firstly, from the point of view of any individual user of the application, some proportion of media data generation, storage and processing now takes place on remote computer systems. Data must be transferred between the nodes of the distributed system over networks that may be unreliable and/or shared with other users and applications. Secondly, a distributed application can easily incorporate multiple simultaneous users, such that there is no longer a single user entity in overall control of the application. Indeed, some components of the system—networks in particular—may be managed by entities that do not participate in the application at all. Users may have differing and conflicting requirements or expectations of application behaviour, especially since they are likely to be using a heterogeneous collection of hardware and software with disparate capabilities and functionality.

These points can be illustrated by a common distributed multimedia application: streaming audio and video over the Internet. Users of these services possess a bewildering variety of hardware running an equally wide range of operating systems. The Internet itself is frequently unreliable and offers no guarantees on the ordering or timely delivery of the data it carries. As a result, the audio and video streams delivered to users are usually of a low quality, with frequent loss of data. This level of performance is unacceptable to corporate users, many of whom are exploring the use of video-conferencing applications as an alternative to expensive face-to-face meetings. In order to provide a high-quality service, suitable for real-time interaction between geographically distant users, these systems typically resort to proprietary hardware and costly dedicated network links. Currently, there is little middle ground between these two extremes, unless the application can be run over a small
area on a high-speed LAN.

1.1.1 Multimedia applications are real-time applications

A distinguishing characteristic of all multimedia applications, distributed or not, is their need for timely delivery and presentation of media data. This is especially true in the case of continuous media streams, where data that is delivered late is generally useless; thus, it is also pointless to request re-delivery of lost data. Multimedia applications are classified as soft real-time systems [KV93] because, while correct application behaviour is dependent on meeting temporal constraints, these may be relaxed—to a certain extent—before the application is considered to have failed. For example, in many conferencing applications it is acceptable for a small proportion of video frames to be lost or delivered late, provided that audio quality is maintained. The particular temporal bounds to be enforced, and their degree of “softness”, are application-specific considerations.

Multimedia applications thus impose significant requirements on the underlying system, both through their real-time nature and their copious resource needs. These demands are only exacerbated when applications are distributed, since then resources must be managed and timing bounds enforced across the whole distributed system, with its additional failure modes and potentially federated management structure. Most widely deployed operating systems and network infrastructures are not capable of fully supporting these applications, for two interrelated reasons: firstly, they do not provide guaranteed reservation of resources such as CPU and network bandwidth; and secondly, even when the necessary resources are available, applications are rarely scheduled appropriately to utilise them within their real-time bounds. Consider a system that is attempting to process two video streams simultaneously, where each stream requires 60% of the system’s total CPU bandwidth to be processed within its timing bounds. A conventional operating system scheduler will attempt to provide a “best effort” sharing of resources between the two streams: they will each receive around 50% of the available CPU bandwidth, such that neither is processed correctly. On the other hand, a system that took account of the real-time and resource needs of the streams would not have allowed the second stream to start once the first was running, or alternatively allowed it to run at a lower quality level, to fit within the available resources.

It is easy to find examples of successful, large-scale, distributed real-time applications; furthermore, many of these systems have been built on top of general-purpose commercial real-time operating systems (RTOS). However, the traditional RTOS application domain is that of embedded systems running, for example, chemical plants, aircraft or home appliances. These applications can generally be exhaustively specified and predictable, deterministic behaviour, so that it is possible to enumerate all possible application states and to engineer the system for the worst case in terms of resource usage or timing constraints. Also, few such applications make use of continuous media data—notable exceptions are the global telephone network and set-top boxes for digital television.
Like their more traditional counterparts, though, these systems too were built to run a single application; their real-time behaviour is again deterministic and predictable.

1.1.2 Multimedia applications are *dynamic*

Beyond their requirement to process continuous media data, distributed multimedia applications differ from traditional real-time systems in another substantial area: the need to be *flexible, adaptable* and *reconfigurable*. As mentioned above, these applications characteristically have multiple independent users with their own particular requirements and expectations of the application. User requirements, and the mix of users themselves, may change over the lifetime of the application; it is reasonable to expect the application to reconfigure to meet these changing circumstances. Likewise, many applications are likely to be distributed over shared network infrastructures, such as some future Internet that supports real-time operation. Applications running in such an environment will compete for resources; also, parts of the infrastructure may fail. In either case, applications should be capable of adapting dynamically to changes in their environment, making the best use of the currently available resources to meet their users' requirements.

Finally, I anticipate that many multimedia systems will be *long-lived*; that is, the lifetime of the *running* application is orders of magnitude longer than the intervals between failures in the underlying system, changes in user mix, or upgrades of application functionality and capabilities. The third point is particularly interesting, as it implies the need to replace application components with new or updated versions, while still maintaining the expected real-time behaviour. Moreover, these reconfigurations must be structured in such a way as to maintain the overall integrity of the application.

It is important to stress that all of these reconfigurations will take place on running applications, so that the changes may affect components that are actively processing real-time media data. Either the reconfiguration is a natural part of the application's ongoing operation, or it is dangerous or costly to shut the application down before reconfiguring it. In any case, if multimedia processing is critical to correct behaviour of the application, then the timing and resource constraints on that processing should be maintained, even while the application undergoes reconfiguration. This requirement is a difficult one to meet. While a great deal of research (see Chapter 2) has addressed the problems of reconfiguring general distributed systems, relatively little has tackled the specific issues of multimedia and real-time systems. It may be that this is because traditional real-time systems have been highly deterministic and devoid of continuous media data, whereas multimedia applications have been developed in flexible—but unpredictable—best-effort environments. To realise the full potential of distributed multimedia, there must be a convergence between these two approaches.
1.1.3 Distributed multimedia requires high-level support

The system support requirements of distributed multimedia extend further than the relatively low-level concerns of resource management and meeting real-time constraints. The applications in question are large and complex, with potentially thousands of users distributed across a wide geographical area. As discussed previously, there is unlikely to be a single entity with overall control over a given application, or the underlying distributed computing platform. Reconfigurations will occur frequently and operate contemporaneously, sometimes with overlapping or conflicting effects. There is even the unpleasant possibility that users will—through ignorance or malice—act to threaten the integrity of the application or the distributed system at large. Yet, despite all of this, mission-critical applications must continue to operate reliably and consistently. Reliable operation of any system, let alone a reconfigurable one, in such a harsh environment poses significant challenges for designers, programmers and maintainers alike. Suitable support mechanisms can ease this burden by relieving individual applications of the need to re-implement common functionality, and by providing a framework that encourages the development of safe, reliable applications.

A support framework should assist an application throughout its lifetime, from its initial instantiation in the distributed system through maintaining steady-state operation to dealing with dynamic reconfigurations. Both development and runtime support are required, and the mechanisms should be generic and flexible enough that they are useful to a large application space, but not so general that the specific needs of distributed continuous-media systems are lost within a bland, all-purpose framework that tries to be all things to all applications. Casting this thought in a more practical light, the framework should support the development of highly programmable applications around a core of basic functionality intrinsic to the target domain. Programmability in this case applies to more than just the application source code: it must also address the need for well-defined abstractions to specify (for example) real-time and QoS characteristics, application integrity constraints and allowed patterns of reconfiguration. These aspects of application behaviour generally cannot be captured conveniently or unambiguously from the source code alone.

In summary, multimedia applications present a unique set of requirements to distributed systems architects, primarily due to the real-time processing constraints of continuous media streams and the dynamic reconfiguration and multiuser interaction that characterises the application domain. Constructing mission-critical systems that meet these requirements poses significant difficulties. I believe that this problem is best attacked through the development of a comprehensive support framework for distributed multimedia systems that provides generic solutions to the shared needs of these applications. The challenge laid before distributed systems and RTOS researchers is to develop generic, highly programmable platforms supporting the dual goals of predictable real-time behaviour alongside flexible, interactive multimedia processing.
1.2 Contributions

This thesis presents a novel approach to the construction and dynamic reconfiguration of distributed, component-based multimedia systems. The underlying theme of the work is the use of a high-level application model to support systems with controlled quality of service running in a resource-managed environment.

The contributions of this work are:

- A set of basic abstractions for the construction of distributed multimedia systems. The architecture seeks to separate the low-level concerns of processing continuous media streams from the more abstract issues of application configuration and management. The key feature of the architecture is the runtime model, a high-level representation of the significant characteristics of the application. This approach is distinguished by the fact that the model is not passive; rather, it plays an active role in the control and management of the application. The model encapsulates:

  - Application structure, both physical (the actual media-processing components and their interconnection) and logical (composition of physical components into more complex components with additional management functionality).

  - Reconfiguration properties. The model stores an application-dependent specification of a valid configuration; the model components themselves implement component-specific reconfiguration actions.

- Techniques to support consistent, atomic reconfiguration of running applications. Atomic actions blend well-known principles of database transactions with continuous media streams, to produce a mechanism for reconfiguration that preserves not only the data carried by continuous media streams, but also the temporal relationships amongst that data. The most important aspect of the reconfiguration mechanism is a scheduling algorithm that manages the tradeoff between temporal integrity, resource utilisation and the timeliness of the reconfiguration in a controlled quality of service environment.

- Validation of the preceding contributions in a prototype implementation of a multimedia programming framework and runtime middleware system. This framework, known as Djin, is a concrete realisation of the preceding contributions and provides a platform for evaluating the research in the context of real-world application scenarios.

\[1\text{The model also incorporates the quality-of-service characteristics of each component; these express the relationships between resource usage and achieved level of service for that component running on a particular hardware/software platform. This part of the research was carried out by Hani Naguib and falls outside the scope of this thesis. For full details, the reader should refer to Mr. Naguib's work in this area [MNCK99, NKMC98] and the statement of conjoint work following the abstract.}\]
1.3 Thesis overview

The remainder of this thesis is organised as follows:

Chapter 2 contains a survey of relevant past research from a variety of areas with a bearing on this work. The topics addressed include: distributed system management, dynamic reconfiguration and adaptation; multimedia programming frameworks and runtime support architectures; the state of the art in audio and video stream coding and switching; and extended transaction processing models. The concluding section of the chapter attempts to draw together these various threads in order to provide some context and motivation for the approach taken in the following chapters.

The original work of the thesis begins in Chapter 3, with an analysis of the peculiar characteristics and requirements of distributed multimedia applications. This is presented through two application scenarios—a security monitoring system and a video switching and editing system for a digital television studio. It should be noted that, as few truly reconfigurable multimedia systems have actually been deployed, it has proved difficult to obtain concrete, real-world application requirements. Thus, the application case studies in this chapter are based on constructed, somewhat hypothetical scenarios, enhanced by input from practitioners who are, or will be, developing similar applications in the near future. The second part of this chapter develops a comprehensive list of requirements for supporting these applications at three different levels: the abstractions provided by a programming framework; a runtime middleware layer implementing these abstractions; and the operating system and network facilities necessary to support the middleware functionality. The derivation of requirements pays particular attention to support for dynamic reconfiguration.

Chapter 4 describes an architecture for composing, controlling and reconfiguring distributed multimedia systems. The first part of the chapter discusses the fundamental high-level concepts of the architecture, while the second part presents the DJINN programming framework and runtime support system that implements the architecture. The most distinctive feature of the architecture is the separation of application components into two distinct layers that enforce a strict separation between the low-level activities of producing, processing and consuming continuous media streams, and the high-level concerns of application structure, management and reconfiguration. Components in the upper layer constitute a dynamic runtime model of the system, encapsulating application structure, quality of service control and reconfiguration functionality. The two layers are entirely separate, so that a failure of the model affects the ability to query and modify the application, but does not negatively impact on the safety of the running system. Details of processing media data on particular hardware, or transporting media streams across particular networks, are confined to the lower layer components.

Chapter 5 deals with the architectural aspects of reconfiguration. A significant advantage of the two-layer model-based approach exemplified by DJINN is that changes to the application can be performed on the runtime model before they are applied to the active media-handling components.
This allows for proposed reconfigurations to be evaluated according to some notion of "correctness" and, if necessary, rejected before they cause irrevocable change to the behaviour of the application. This chapter develops a formal definition of reconfiguration and introduces the concept of atomic actions, a reconfiguration mechanism derived from traditional database transactions and sharing their properties of atomicity and consistency. Atomic actions are validated against system-and application-defined consistency constraints before they are committed, where consistency covers both application structure and quality of service guarantees.

Following directly from the discussion of atomic actions, Chapter 6 addresses the problems inherent in reconfiguring active continuous media streams. Because components may continue to handle media data even while they are being reconfigured, a naïve approach to the commit phase of atomic actions can easily lead to loss of data or violations of the temporal relationships within and between streams, either of which may be treated as incorrect application behaviour. This chapter presents an algorithm that endeavours to reduce and ideally eliminate such undesirable states through appropriate scheduling of the updates to active media-handling components. In its current form the algorithm is generic for certain defined classes of applications.

Chapter 7 presents an empirical evaluation of the architecture developed over the preceding three chapters, along with its prototype implementation in the DJINN framework. A series of experimental trials were run on a test application derived from the remote surveillance system introduced in Chapter 3. The experiments aimed to provide some validation of my approach to dynamic reconfiguration by comparing the performance of the test application under different variations of the reconfiguration scheduling algorithm, versus atomic actions with no scheduling support. This chapter includes summarised results of the experimental trials and a discussion of their implications for the design of the reconfiguration architecture and the DJINN framework. Additional results are presented in Appendix A.

Chapter 8 concludes with a summary of the work presented in the rest of this dissertation, and a review of its aims and contributions. The work is assessed with respect to whether each of the stated goals of the architecture has been achieved or not. This chapter also lays out some directions for future research.
CHAPTER 2

RELATED WORK

This chapter presents a survey of current and recent research that is relevant to the work described in this dissertation. The survey addresses three main areas of research:

- Firstly, Section 2.1 examines the state-of-the-art in programming frameworks for multimedia.
- Section 2.2 provides an overview of current support for dynamic reconfiguration of general distributed systems.
- Finally, Section 2.3 discusses available mechanisms for adaptation and other approaches to dynamic reconfiguration in distributed multimedia systems.

The reader should be aware that there is a degree of overlap between these categories; many of the systems surveyed could as easily appear in one section as another. Furthermore, this is by no means an exhaustive survey; rather, it discusses the work that has had the most influence on the research, design and implementation presented in the remainder of this thesis.

2.1 Multimedia programming frameworks

Before the mid-1990's, multimedia systems were generally constructed—outside the research community at least—on an ad-hoc basis, were usually targeted at a specific hardware platform, and rarely supported any kind of distributed operation. Operating system support, where it existed at all, tended to be at the level of media devices, for example Microsoft’s MC1 (Media Control Interface) library provided a generic control interface for devices such as laser disc players, tape machines and CD-ROM drives. One entirely predictable result of this approach is that each new application tends to reinvent more than one wheel, leading to extra work for the developers and often sub-optimal solutions.
Thus, it seems reasonable to provide generic support for the shared properties of multimedia systems. Such support must address not only programming issues—provision of abstractions and interfaces for common multimedia programming tasks, for instance—but also the realisation of those abstractions at runtime. This section reviews a number of systems that attempt to meet this requirement.

2.1.1 MIT VuSystem

Researchers in the Telemedia Networks and Systems Group at the Massachusetts Institute of Technology (MIT) developed the VuSystem [LWT94, LT96], a programming system for the construction of what the authors describe as computer-participative multimedia systems. The term refers to applications that perform analysis on incoming multimedia data streams, then take actions based on this analysis. Examples of such applications include face recognition and content-based storage and retrieval of television news broadcasts. This is in contrast to computer-mediated multimedia systems, where the computing infrastructure simply transports multimedia data between users without regard for the content; a video-conferencing system, for example.

All VuSystem applications are built on a two-layer architecture that splits the application code into in-band and out-of-band processing partitions (Figure 2.1). The out-of-band partition is essentially the familiar event loop common to most interactive programs; it performs event handling (including user input events) and other high-level application-specific processing. Out-of-band and user interface code is written as Tcl scripts [Ous94] and uses a custom X-Windows widget set rather than the Tk user interface toolkit. In-band processing is then the repetitive, time-sensitive computation performed on each element of a media data stream. The in-band partition is structured as a reconfigurable directed graph of processing modules, written in C++.
In-band architecture: modules and payloads

Modules are classified according to the number of input and output ports they possess; associations between ports define the edges of the in-band processing graph. A large number of modules have been written, including filters for image processing and JPEG video coding.

Once connected, modules communicate by passing media data payloads through their ports. Payloads are self-identifying, dynamically-typed units of media data, each of which might contain, for example, a single frame of video or a sequence of audio samples. A payload object has two components: a descriptor and an opaque chunk of data. The descriptor component includes general information common to every payload, as well as type-specific data that also serve to identify the precise representation used by the data component. Descriptors are implemented as sets of C++ member functions and include methods to encode and decode themselves to and from byte streams, thus allowing payloads to be transmitted across a network or stored in a file. Data components are always stored in shared memory segments to facilitate cheap (local) interprocess communication.

Communication between modules is regulated by the module data protocol, which specifies a mechanism for transferring payload ownership between an upstream and a downstream module. The protocol is very simple: a downstream module may either accept a payload or reject it. In the former case, the upstream module will clear any references it holds to the payload while in the latter it will retain an internal reference until the data has been successfully transferred. The "back-pressure" resulting from rejected payloads allows a crude notion of application timing constraints to propagate back towards the source(s) of the module graph.

Scheduling and synchronisation

The VuSystem scheduler is designed to run all VuSystem code on a single thread, with no preemption. This approach simplifies the implementation of modules by eliminating the need for locking or critical sections in the code. The scheduler is tightly integrated with the user-mode scheduler provided by the X-Window system toolkit, Xt, allowing combined scheduling of in-band media processing and out-of-band user interface activity.

CPU resource scheduling relates directly to the flow of payloads through the system. Input-processing modules are only scheduled when they receive new data; likewise, output-processing modules will be scheduled only after their data has been accepted by a downstream module. Thus, payloads effectively becoming scheduling tokens, controlling the allocation of processing resources to the modules they pass through. Media streams will automatically adapt their rates to reflect the changing availability of CPU cycles, without any intervention from the application itself.

In order to support synchronised capture and display of multiple media streams, payloads are timestamped at creation time. These timestamps will be used to determine the absolute presentation
time for each payload when it is processed by a sink (a special filter module adds an offset to each
timestamp allowing a stream to be replayed at a later time than it was recorded at). As a means of
synchronising multiple streams, this technique is heavily reliant on tight synchronisation of clocks
between source and sink devices.

Out-of-band interface

VuSystem programs run within the application shell—an extended Tcl interpreter statically linked
with the set of C++ modules required by the current application. In-band modules are created, de-
stroyed and configured using standard Tcl commands. Modules can communicate asynchronously
with the out-of-band code through the use of callback functions that pass events into the Tcl code.
The execution of callbacks will generally be delayed by the scheduler to prevent interference with
time-critical in-band code.

Other work by the same group at MIT has led to the development of the ViewStation architecture
[LWS+94]. The ViewStation is a distributed multimedia system built around a gigabit local-area
ATM network and targeted at live video-processing applications. The authors argue that complex
congestion-control and resource management functions are not necessary in a local environment
provided that the network infrastructure has a limited number of access ports and bandwidth signi-
nificantly greater than that required by its clients.

The software architecture of the ViewStation is based on the VuSystem. A variety of applications
have been implemented, including room and whiteboard monitors and systems for content-based
retrieval of broadcast news and sports. A related project developed has extensions to the original
single-threaded VuSystem to support programming and execution of distributed applications.

2.1.2 Berkeley CMT

The design of the Berkeley Continuous Media Toolkit (CMT) [MPR97], developed by the Multi-
timedia Research Center (BMRC) in the University of California at Berkeley (UCB), was clearly
influenced by the earlier VuSystem work at UCB. The CMT is just one part of a much larger body
of research conducted by the BMRC, including work on MPEG video compression [RPSL94], media
storage, video-on-demand and local-area video broadcasting.

Like the VuSystem, the CMT offers a two-layer approach to application construction. At the top
level, application logic and user interface behaviour are implemented as Tcl scripts. The Tcl inter-
preter is extended by the Tk graphical interface toolkit [Ous91] and a set of distributed programming
programming abstractions known as Tcl-DP [SRY93]. The second level of the system is the CMT li-
brary layer, consisting of a collection of multimedia processing component types implemented in C
and compiled in as extensions to the Tcl interpreter. As with the VuSystem, the fact that the media
processing code is linked directly with the Tcl interpreter binary means that it is difficult to add new
functionality to the platform without recompiling, then restarting any affected applications. The choice of C as the implementation language for the library is an unfortunate historical artifact, robbing the CMT of the object-oriented programming abstractions available to the VuSystem—such as the ability to define an abstract base class for all audio decoders, then implement a generic application that will work with any audio decoder that inherits from the base class. Coupled with the fact that CMT components cannot be composed, the architecture tends towards monolithic components, such as an MPEG decoder that also display the frames after decoding.

Application code layer

The Tcl-DP distributed programming abstractions permit client/server and other distributed application structures to be built by a Tcl script. Essentially, the Tcl-DP API enables a programmer to make a connection to a remote Tcl interpreter process, then invoke operations on objects within that process via an RPC mechanism, as though they were local objects. [MPR97] discusses three different application models:

1. **Basic VuSystem-style application.** All media processing components run in a single CMT process and are controlled by a Tcl script running in the same process.

2. **Simple distributed application.** A “master” CMT process creates connections to one or more existing remote CMT processes; it is then able to create and manipulate CMT objects in those processes.

3. **Complex distributed application.** Here the master process is a standard Tcl-DP interpreter; that is, it contains no media processing component code. As in the previous model, the master process manages the operation of CMT objects in one or more remote CMT processes, but does not itself play any part in the processing activity of the application.

CMT library layer

CMT objects “operate with the basic premise of receiving media-specific data from either a device...or another object, operating on the data in some way, and then sending the data to either a device...or another object.” As well as the source, sink and filter objects of the VuSystem, CMT adds a fourth classification, the transport pair. Transport pairs are responsible for sending and receiving media data across a network and typically come in matched pairs, such as UDP send and receive objects.

Berkeley CMT extends the producer-driven data transfer model of the VuSystem to support both “push” and “pull” styles of communication. Push transfers are roughly analogous to the VuSystem approach, while in the pull model media data transfer is initiated by the consumer object invoking a method on the producer. The exact method called by the producer or consumer of data in the
push or pull model, respectively, is determined by the programmer and can be modified at runtime. These methods may be arbitrary Tcl commands, allowing the easy creation of multiplexer or "mixer" components. The CMT architecture does not appear to offer any direct equivalent to the back-pressure rate control mechanism used by the VuSystem.

Media data are stored in buffers implemented as shared memory segments. The CMT defines the semantics of buffers in considerably more detail than the VuSystem: a buffer manager API offers services for buffer creation, use, reuse and destruction to both application programmers and CMT library objects. A particularly useful feature is the notion of scatter-gather buffers which allow a series of disjoint memory segments to be treated as a single continuous region, thus avoiding the need to allocate and copy large areas of memory to perform simple tasks such as prepending a TCP/IP header structure to a media data element.

Logical time and event handling

The CMT also offers richer support for media scheduling and synchronisation that the VuSystem. A CMT Logical Time System (LTS) is a linear mapping between system time—itself only an approximation to real time—and an infinite logical timeline. It allows an application to maintain its own meaningful concept of the passage of time with respect to the media streams it processes. Multiple LTSs can exist and be synchronised with one another, even across host boundaries. Each unit of continuous media data is mapped to a block of logical time in some LTS, and applications schedule actions in the real world by converting logical time values back to system time. Because each media element can be mapped independently, the system itself makes no assumptions about the timing relationship between elements of a stream; such decisions are left to applications.

Several event handling services are provided by the CMT library layer. The at queue allows media processing objects to schedule a callback (on themselves) within a particular time range. A second callback may be specified, to be called if the first callback could not be made within its time constraints. The priority groups mechanism allows the programmer to set the relative priorities of different groups of objects to, for example, indicate that audio processing was more important than video in a particular application. Finally, a facility exists to bind continuous media events to actions defined as Tcl commands. Supported media events include playing, receiving and dropping a media data element.

Related work at UC Berkeley

While the Berkeley CMT is a significant achievement, the system has a number of architectural drawbacks as mentioned above—in particular the use of a non object-oriented implementation language and the resulting flat, monolithic component structure. The BMRC researchers recognise these limitations and have set about creating a new system—the mash toolkit—that retains the best
features of the VuSystem, the CMT and the MBone conferencing tools [Hol95, MJ95], also partly developed at UCB. The goal of the mash project is to develop a "fine-grained, extensible and high-performance toolkit" that provides a common programming infrastructure for future multimedia networking research efforts.

The mash architecture [MBK+97] shares the same two-layer Tcl/C++ approach of the VuSystem. However, the Tcl layer has been significantly enhanced by the addition of the O’Tcl extension [WL95] that provides true object-oriented programming abstractions within Tcl. A mash object is "an abstract entity whose methods can be implemented on either side of the O’Tcl/C++ boundary." Complex, application-specific abstractions can be constructed by "coalescing" (composing) simpler objects together using O’Tcl scripts. The authors of [MBK+97] are careful to justify their split-level approach on the grounds that policy and mechanism are inherently different and should be implemented in different languages suited for each purpose. The mash programming model is almost completely event-driven, with significant use of timers to trigger periodic events such as video frame capture.

2.1.3 Gibbs and Tschritzis

The multimedia framework developed by Gibbs and Tschritzis (the "G&T" framework) [GBD+91, Gib92, GT95] uses the now familiar component and port abstractions to construct applications, but goes far beyond this basic approach in its comprehensive modelling of multimedia data types and operations. In [GT95] the authors state their view that the object-oriented programming paradigm is the best way forward for multimedia programming:

"Object-oriented multimedia seems almost inevitable... There appears to be a natural fit between multimedia programming on one hand and object-oriented programming languages on the other."

To justify this view, the framework defines an exhaustive hierarchy of both static and temporal media classes, modelling the elements of media streams as they are processed by the framework components. There is a parallel hierarchy of format classes that encapsulate the same media representations on storage devices or over communications links. A third hierarchy of transform classes represent processing operations that can be performed on media data.

G&T components are semi-autonomous objects that encapsulate time-critical operations on media streams. Input and output ports are not connected directly together as in previous frameworks; rather, each inter-component connection is mediated by a connector object. Connectors have their own independent class hierarchy entirely separate from that of components. It is unclear why the component “processing” and connector “transport” activities have been so completely divorced in this way, given that the transfer of media data across a network, for example, could be considered
just as much of a time-critical operation as any computation on the data itself. G&T supports
distributed applications through the use of specialised network connector classes, but the authors
do not go into detail on how such an application would be programmed, instantiated and managed.
The framework does not itself offer any mechanism for component composition or coalescing as
found in the VuSystem or mash; however, composite components can be constructed on an ad-hoc
basis with minimal programmer effort.

G&T's most important contribution lies in its detailed and thorough specification of the framework
of abstract classes covering all aspects of a multimedia system, from querying properties of media
formats through to timing control of components. In contrast to the systems discussed previously,
G&T applications can be considered as being rather strongly typed: it is very difficult to construct
an application that is inconsistent, in terms of its port interconnections and the media types handled
by its components, whereas such errors of "faulty plumbing" are easy to make—yet relatively hard
to track down—in a loosely typed framework based on the VuSystem model.

2.1.4 Medusa and Pandora

The Medusa multimedia environment [WGH94] was developed by researchers at Olivetti Research
and the University of Cambridge Computer Laboratory. It takes much of its inspiration from the
earlier work on Pandora [JH93]. Like the G&T framework, Medusa uses a peer-to-peer, component-
based paradigm for constructing applications. However, in contrast to the other systems discussed
above, Medusa also specifies the particular hardware and network platform on which it is to run.
The architecture is based around the notion of intelligent "ATM direct peripherals". These are
essentially small, dedicated computers running an embedded real-time operating system and con-
necting some I/O device to the ATM [LMT93, Par94] network. All communication takes place over
the ATM network, allowing for scenarios such as a direct ATM connection between a video camera
and a window display without any intervening processing stage in a workstation.

Medusa's components are active objects that communicate through dynamic collections of named
ports. While there is no direct support for composition, in the sense of absorbing several simple com-
ponents into a larger, more complex one, it is possible to configure arbitrary chains of proxy objects
between an application module and the low-level media handling modules. The literature mainly
discusses proxies in the context of providing security for other modules, but it appears feasible to
develop "smart" proxies that could simulate the behaviour of true module composition.

The most important contribution of Medusa lies in the details of the inter-module communica-
tion mechanism. Because it is assumed that all Medusa applications will run in an ATM environ-
ment, certain simplifying assumptions can be made about the underlying network. Firstly, there
is a global addressing scheme, so any module can be connected to any other. More importantly,
network connections are assumed to be simple, cheap and plentiful—which is certainly true in an
ATM environment. With these assumptions in place, the designers elected to make all inter-module
connections reliable by default. In general there are no explicit connector objects—connections are simply associations between endpoints at the ATM level. If different semantics are required for a connection, or a connection needs to be tunnelled across a non-ATM link, a connection buffer module is used. A connection buffer might be used to provide buffering or to manage quality of service (QoS) between two modules, or it may be optimised for a particular kind of media data. Unlike the independent connector objects found in the G&T framework, Medusa connection buffers are themselves modules, as is the overall controlling application.

The actual transfer of continuous media data between Medusa modules is synchronous and demand-driven, that is, data is sent by a producer upon receipt of a request from a downstream consumer. In an ATM network the cost of sending data requests is relatively low, but this is not necessarily true for other network environments. Since ATM is itself able to provide some limited latency and throughput guarantees, it seems that an asynchronous, rate-based transfer model—with a feedback mechanism to trigger rate adjustment—would be more suitable for this environment.

The goal of the earlier Pandora architecture was to "...create the architecture for a general-purpose distributed multimedia system and build an operating system for it that supports multimedia applications." Central to the system is the idea of a multimedia workstation which consists of a conventional workstation and various multimedia I/O devices, all connected to a local ATM switch; these local clusters are in turn connected to a wider ATM network. The role of the workstation is largely restricted to control of the local switch. Whereas Medusa's peripherals are in essence complete embedded computer systems, I/O devices in Pandora are not always as flexible. For example, the Pandora display device inserts incoming video "tiles" directly into the display using analogue switching techniques. Pandora does not provide any high-level support for multimedia application programming. Another system that uses a similar ATM-based architecture is Pegasus [MLM94], developed at the Universities of Cambridge and Twente. Pegasus adds distributed multimedia support to the Nemesys microkernel operating system [Hyd94, Ros95, LMB97].

2.1.5 Java Media Framework

The Java Media Framework (JMF) is a product of Sun Microsystems, in collaboration with Silicon Graphics and Intel. The framework consists of an API and set of libraries supporting continuous media streaming within Java [AGH00, JSG88] programs. The JMF is one part of a larger effort by Sun to integrate multimedia functionality with the Java environment [Sun]. Early versions of the JMF [JMF98] only provided support for media playback—although including streaming from remote sources—while the latest releases [JMF99] have added media capture and filtering capabilities. The software is available in "native code" editions for particular hardware and operating system platforms, as well as a more portable but poorer-performing pure Java version.
Related Work

![Diagram of JMF media player states](image)

Figure 2.2: JMF media player states (from [JMF98]).

Media player APIs

According to [JMF98], the main purpose of the JMF media player APIs is:

"…to support the delivery of synchronized media data and to allow integration with the underlying platform’s native environment and Java’s core packages. The Player APIs support both client pull protocols, such as HTTP, and server push protocols, such as RTP."

A player object processes a stream of continuous media data obtained from a data source object, rendering or presenting each element of the stream at the appropriate time. The data source itself completely encapsulates the real, possibly remote, location of the stream source and the mechanism used to deliver the data to the player. As per the above quote, the various data source subclasses operate in either client-driven ("pull") or server-driven ("push") mode. A data source object is specific to the source stream it encapsulates; it cannot be used to retrieve data from another source.

The JMF makes full use of Java’s class inheritance and interface implementation capabilities to provide a rich set of semantics in media player objects. In particular, a player exists in one of six well-defined states (Figure 2.2), ranging from Unrealized, where the player has been instantiated but does not yet know anything about the media stream, through to Started where the player is fully configured and running. In between are four stages of player realisation and prefetching, during which the player acquires any local and remote resources it requires and optionally preloads some initial data from the stream to reduce startup latency and jitter when the presentation begins. State transitions are initiated by method invocations on the player; these methods run asynchronously and their completion is indicated by the delivery of events through Java’s standard non-distributed event mechanism.

The JMF approach to media timing and synchronisation is broadly similar to that used by the Berkeley CMT. The JMF runtime maintains a master timebase object—a monotonically increasing counter synchronised with the system clock. It appears to be possible to create other timebase
objects derived from the master timebase, although the documentation is not entirely clear on this point. Each player object is associated with a timebase and also maintains its own notion of media time—the current point with the player’s media stream. Media time is non-linear and may be modified in arbitrary ways. A clock object in each player defines the mapping between the player’s media time and its timebase. Synchronisation of streams is achieved by associating them with the same timebase.

Capture and filtering APIs

Version 2.0 of the JMF [JMP99] is a significant upgrade to the framework. It retains the same high-level application model as the previous versions—maintaining compatibility with existing applications—and extends this to support media capture and intermediate processing. In addition, a second, lower level to the application model is specified, support custom media processing and extensions to the standard framework. The new framework also defines a much more detailed hierarchy of media types and a wider range of data source subclasses.

Media capture is implemented through the existing data source abstraction. That is, a data source object may be bound directly to a capture device such as a camera or microphone as well as the more abstract media sources supported previously. Such a capture source can then be embedded within a player object; the player will retrieve and present the captured data in the usual way. Furthermore, a new class of processor objects is introduced. A processor takes input from a data source—in the same manner as a media player—processes the data in some way, then sends the modified data to a presentation device or a second data source. Processors are thus similar to players but without the final presentation stage; instead, the data is passed to a “downstream” object. The use of processor objects allows arbitrary media processing sequences to be constructed, analogous to the component pipelines of the VuSystem or Berkeley CMT. The difference is, of course, that in the component-based models the structure of the pipeline is made explicit by the connections between input and output ports, while in the JMF objects are simply embedded within one another.

2.1.6 Microsoft DirectShow

The JMF is intimately bound to the Java programming environment and even to particular features in Sun's standard Java class libraries. Similarly there exist other multimedia frameworks that are just as tightly integrated with particular operating systems. One example of such a framework is Microsoft DirectShow [MSDb], the media streaming architecture found in recent versions of the Microsoft Windows operating system. DirectShow replaces an earlier series of streaming APIs including Video for Windows (VFW) and ActiveMovie [MSA]. It forms part of Windows' DirectX [MSDz, MSDe] multimedia subsystem which provides controlled, low-level access to multimedia hardware and a range of other media-processing APIs such as Direct3D, DirectDraw and Direct-
Sound. The architecture is based around Microsoft's Common Object Model (COM) [MSC]. Despite the availability of Distributed COM (DCOM), DirectShow does not directly support distributed applications; rather, various third parties have provided components that can obtain remote stream data from, say, an RTP [SCF96, Sch96] source and deliver it to an otherwise non-distributed DirectShow application.

DirectShow applications utilise the ubiquitous directed graph of components model, except that the documentation refers to filters and pins rather than the components and ports of most other frameworks; the basic concepts are the same. Applications use a filter graph manager object to construct and operate DirectShow component pipelines. The filter graph manager encapsulates some powerful functionality: it is able to search for a filter graph configuration that can render a particular media type and automatically construct the appropriate graph. Details of all the available filter components are stored in the Windows registry, a database of hardware and software configuration data maintained by every Windows-based computer. Since, like all COM objects, the filters are implemented as DLLs (Dynamic Link Libraries; equivalent to shared objects on a UNIX system) it is trivial to add new filter types to a running system—contrast this with the VuSystem or Berkeley CMT where the entire functionality is contained within a single Tel interpreter binary.

A DirectShow application is able to take full advantage of Windows' multithreaded execution model. The programmer is free to choose an appropriate level of multithreading and to manage the transfer of control between different threads and filters. Rate control—adapting to changing system load—is achieved by filters sending quality control messages to their upstream peers. If a filter is unable to process the amount of data it is receiving, it requests that the upstream region of the filter graph send less; conversely, if a filter determines that it could safely handle a higher data rate, it will ask the upstream filters to send more. The net effect is similar to that realised by the back-pressure mechanism of the VuSystem.

2.1.7 CORBA audio/video streaming specification

As CORBA technology [OMG99] becomes more widely adopted as a means for building interoperable distributed systems, it is inevitable that applications will come to incorporate both traditional invocation-based CORBA interactions and multimedia streaming functionality. In an attempt to meet the need for multimedia support within the CORBA framework, the OMG has released a formal specification of an audio/video streaming service [OMG00]. The final specification was developed from proposals received in 1996 and has remained essentially unchanged since 1998.

The OMG specification proposes a set of interfaces that implement a distributed multimedia streaming framework. Continuous media data transfer is represented by a stream where a stream may itself contain multiple data flows that can be controlled individually or collectively via operations on the stream. Separate flows may travel in both directions within a stream. Multimedia hardware and
software objects—stored media files, for example—are encapsulated by multimedia devices. A multimedia device object may encapsulate more than one physical or logical device and can support multiple streams simultaneously. For each application-level stream connection using a given multimedia device, the device object will create a virtual multimedia device and a stream endpoint which represent the device-specific and network-specific aspects of the stream endpoint respectively.

A stream endpoint logically encapsulates the endpoints of each flow within the stream. Two distinct flavours of stream endpoint exist: “A-party” and “B-party” endpoints. Either type can contain any combination of producer and consumer flow endpoints. However, binding (connection) is only supported between an “A” and a “B” endpoint, so that “...when an instance of a typed StreamEndPoint is created the flows will be plumbed in the right direction... The choice of which endpoint is the A party and which is the B party is entirely arbitrary.” [OMG00]. Static checking of endpoint classes is employed at compile time to ensure that only compatible endpoints are connected. This simple scheme based on subclassing does not address all of the issues involved in determining compatibility between stream interfaces: for example, it is perfectly feasible for an application to want to connect just the audio output of a combined audio/video source to an audio-only consumer; however, static type checking would determine that the two interfaces were incompatible. It is possible to set up such a configuration by direct connection of individual flows, although this requires far greater programmer effort and nullifies any advantage gained by bundling multiple flows into a single stream.

Stream control

The stream control interface abstracts the continuous media transfer between virtual multimedia devices. It provides operations to initiate binding between endpoints—although the binding process is managed by the endpoints themselves—and to start and stop data flow within the stream. More complex operations such as pausing or rewinding a stream are not supported by the base stream control interface, although they may be implemented by subclasses. While this is arguably a serious omission in the specification, it should be noted that multimedia devices do exist that cannot support any operations beyond simple start and stop actions—consider a device encapsulating receipt of a multicast stream from a remote, non CORBA-compliant source, for example. Thus, I contend that the OMG has taken the correct approach in keeping the stream control interface as simple as possible. However, the specification would benefit from the inclusion of some well-defined generic media control interfaces that would be implemented by devices or stream control objects requiring extended functionality.

2.1.8 Discussion

This section has explored the state of the art in programming support for distributed multimedia applications, covering a range of research- and commercially-oriented multimedia middleware plat-
forms. Almost without exception, these systems adopt a component-based application architecture: media processing functionality is encapsulated by component objects that produce and consume media data through ports. Inter-port connections permit the construction of continuous media processing "pipelines". Two of the surveyed systems deviate somewhat from this model. In the IMF, connectivity between components is not explicitly represented; rather, components are generally "connected" by embedding one as an instance variable of the other. Overall, the effect is the same, but this mechanism does not encourage runtime reconfiguration of the component pipeline. The CORBA audio/video streaming specification is based around the idea of stream interfaces: a typed stream object encapsulates the continuous media flow between two devices. This model does not directly support the creation of arbitrary processing pipelines.

It is interesting to contrast each framework's approach to communication between distributed components. Several systems—G&T, Medusa and CORBA—represent this communication explicitly via objects in the component graph. These connector objects typically export a control interface allowing the application to modify selected aspects of the communication. On the other hand, the VuSystem-derived platforms, and the commercial packages, rely on components that simply know how to send or receive media data to or from the network; there is no explicit link between the distributed segments of the component graph.

The VuSystem, Berkeley CMT and mad all utilise a split-level application architecture, with Tcl scripts used to construct and manage the operation of the underlying component object graph. This approach provides a useful separation of concerns between component implementation and application structure. DirectShow offers a similar approach, with all component operations mediated by the COM object management layer and components only accessible through their declared interfaces. Along with G&T, DirectShow is also one of the few frameworks to offer the programmer complete control over threading, scheduling and control transfer. IMF components can in theory make use of Java's threading and concurrency-control APIs, but this behaviour is hidden by the implementation and cannot be modified by the programmer. The same two platforms—DirectShow and G&T—are the most strongly type-checked of the systems surveyed here. Connections between components are validated dynamically, at runtime, and the connection cannot be made unless the media types on either side are compatible. Type-checking at any level means that components do not have to check every media element as it arrives, while performing the checks at runtime encourages extensibility. CORBA also implements types-checking of stream connections, but only statically, at compile time.

None of these frameworks provides direct support for dynamic reconfiguration. The VuSystem-derived platforms certainly support runtime manipulation of the object graph through their Tcl scripting interface. However, this ability is hampered by the fact that all of the desired component types must be statically known at compile time—there is no facility to dynamically load new, previously unknown component classes at runtime. IMF and DirectShow both offer a suitable dynamic loading facility; IMF through Java's standard class loader and reflection functionality, and Direct-
Show through the combination of Windows DLLs and the abstraction provided by the COM layer. Sadly, both systems proceed to squander this ability by offering no means to reconfigure their media processing structures after initial instantiation.

2.2 Distributed system composition and reconfiguration

This section examines a selection of methodologies and systems supporting dynamic adaptation and reconfiguration of distributed applications, ranging from the relatively low-level proxying scheme implemented for Equus to the language-based approach of Darwin or the SIRAC project. None of the work discussed here specifically addresses the problems inherent in reconfiguring multimedia systems; in particular the real-time constraints imposed by the application domain and the consequent requirement that media stream data continues to flow even while the application is being reconfigured.

2.2.1 Client-server reconfiguration in Equus

Research carried out at the University of Westminster and Queen Mary and Westfield College (QMW), University of London, and presented in [Kin93], describes a method for dynamic reconfiguration of groups of server processes, based on the use of proxy processes acting as intermediaries between the servers and their clients. The mechanism has been implemented in the Equus distributed operating environment [Kin90].

The system is introduced in the context of reconfiguring general distributed computations, an area to which much research effort has been devoted, such as [LF85, Kra94, TKK+94, GSM+95, SK96, Bec92, WS96, WL96, SW98, We98]. A generally accepted model for such reconfigurations is that the target system to be reconfigured consists of a set of independent processes interconnected by message-passing communication links, along with a mapping of this structure onto the underlying processing and network hardware—alogous to the component pipelines used by the multimedia frameworks in the previous section. Kindberg's work is particularly concerned with the dynamic reconfiguration of server software used by clients in an open distributed system. Clients and servers in such systems are relatively loosely coupled, and can make fewer assumptions about each other's behaviour than the processes in a tightly coupled, closed systems. Thus, it is desirable that reconfigurations be transparent to clients, and that servers should continue to present a consistent view of the provided service before and after a reconfiguration.

Interaction between clients and servers often takes the form of a transaction, as defined by [KM90] and [Kin93]:

"[A] transaction is a series of bilateral processing steps involving message exchanges which
leaves both client (the sender of the first message) and server in a mutually consistent state, defined relative to the application and service."

Furthermore, a series of such transactions might be necessary in order to leave a collection of processes in a consistent state. Given that reconfigurations will often involve replacement of existing server processes—and require state to be copied from each old process to its replacement—it is important that the system be in a state where its integrity cannot be affected by the reconfiguration; that is, none of the reconfigured process can be currently engaged in transactions. Therefore, the reconfiguration mechanism must be able to detect, or infer, the beginning and completion of transactions possibly covering many processes. The Kindberg architecture does not assume the presence of explicit beginTransaction and endTransaction messages; rather it asserts that transaction boundaries may be inferred from the data passed between processes, as long as transactions that overlap in time do not also overlap in the data items they exchange.\footnote{Timeouts are also used to infer that a transaction has completed, where there is a known upper bound on the duration of a given interaction.}

The architecture aims to be able to effect reconfigurations without any participation from the processes involved, thus it must provide means to bring the system into a suitable state for safe reconfiguration whilst client processes continue to perform their normal activities. This goal is achieved through the use of special proxy processes, derived from those used by Shapiro in [Sha86], and placed between clients and servers. The proxy intercepts each message passing between a given client and server, and decides whether to deliver the message immediately or buffer it. Clients and servers are not aware that a proxy is mediating their communication. A state of server-quiessence is defined, whereby a server \( S \) is said to be server-quiessent with respect to a client \( C \) if it is not engaged in any transactions initiated by the client and will not receive any subsequent messages from the client relating to new transactions. In essence, such a state may be obtained by the proxy between \( S \) and \( C \) buffering any messages initiating new transactions, and waiting for all outstanding transactions to complete. Then, in order to transparently associate \( C \) with a new server \( S' \) it is sufficient to copy the relevant state from \( S \) to \( S' \) and instruct the proxy that messages from \( C \) should now be forwarded to \( S' \) rather than \( S \). Certain complications arise if transactions that completed before the reconfiguration have initiated other transactions with the potential to affect the state of \( S \) via clients other than \( C \); the system introduces additional assumptions on the overlapping and synchronisation of transactions to deal with these cases. A second algorithm is also presented, supporting reconfigurations that involve the complete removal of clients and/or servers. This invokes additional notions of client-quiessence and passivity, following [KM90].

Potentially the biggest problem with the approach taken by this work—in its rôle as a tool for reconfiguring open distributed systems—is the number of assumptions made about the nature of the interaction between clients, servers and other entities in the system. Even if we discount the possibility of clients and servers communicating via channels outside the message passing mechanism,
it is highly unlikely that all messages will be in a form that a generic proxy is able to reason about. Similarly, it seems improbable that a general-purpose reconfiguration mechanism would be able to identify and copy server state relevant to a particular client without considerable help from the servers in question. To be fair, the author admits that some of these limitations exist and proposes some directions in which to search for solutions. Nevertheless, the work is particularly relevant to this thesis, as it deals with the reconfiguration of loosely coupled systems where individual components have minimal knowledge of their peers’ identities and functionality. Likewise, the idea of ensuring a consistent state for safe reconfiguration, and the use of transactions, will prove to be important in subsequent chapters.

2.2.2 Self-adaptive software for DSP applications

The design of digital signal processing (DSP) systems is tightly bound to the nature of the signal source and any environmental noise that must be accounted for. Should operating conditions change such that the design assumptions of a DSP system are no longer valid, the DSP algorithm must be modified to reflect the new environment. It is impractical to design a single algorithm that can cope with all conceivable environmental change; thus, adaptive systems are generally preferred. Parameter adaptive systems are relatively widespread; in these, the DSP algorithm has a fixed structure whose operation is parameterised. However, a fixed structure is known to have limitations in rapidly changing or unstable environments. In contrast, a structurally adaptive system is able to modify its own structure—the DSP signal flow—while it is running. In \cite{Kar95, SKB95, SKB98}, Sztipanovits et al. describe a methodology for constructing and reconfiguring such systems.

DSP systems are typically represented as signal-flow graphs, where the nodes of the graph represent computational elements and the edges the signal flows between them. In addition, signal processing graphs may be hierarchically composed in a similar manner to the media processing graphs of, say, mash. Graphs are translated into applications by compilation or direct runtime execution of the graph. In either case, the translation must determine a schedule for the operations to be performed on the signals flowing through the system. The production of a running application from a domain-specific structure such as a DSP signal-flow graph represents an instance of model-integrated computing, where high-level abstract models are used in building and analysing a system. The models provide sufficient information to synthesise the full application, and provide a means of analysis: in the DSP case, the timing behaviour of the application can be derived directly from the signal-flow graph.

System architecture

Sztipanovits, et al. have developed the Multigraph Architecture (MGA) to support model-integrated computing in a variety of domains, including DSP design. A key feature of this architecture is the
introduction of an intermediate level of abstraction—the Control Graph—between the very abstract representation of the system and its executable realisation. The control graph exists at runtime and is used to control the scheduling of activity in the compiled signal processing elements; it also plays an important rôle in dynamic reconfiguration. Runtime support for the MGA is provided by the Multiograph Kernel (MGK) which supports the creation and execution of signal flow graphs. A graph is run by a scheduler operating on dataflow principles; that is, processing elements are activated when data is available at their inputs.

The control graph itself is constructed by a further system component, the Model Interpreter, that performs the mapping between the abstract, high-level model of the DSP system and the executable model embodied in the control graph, which can be run by the MGK. The model interpreter operates by expanding the top-level, hierarchical signal flow model into a tree of interconnected builder objects, where the leaves of the tree correspond to runtime computational elements, and are responsible for instantiating these objects. Once the builder objects have done their work of creating the runtime processing network, they can in principle be discarded. However, if the builder object tree is retained, it can be used to implement dynamic reconfiguration, allowing the computational structure of the DSP application to be changed while it is running.

Dynamic reconfiguration

The reconfiguration mechanism assumes that the target DSP system has a finite number of possible operating modes, each represented by a well-defined signal flow graph structure. Thus, any compound object in the hierarchical system model may contain a number of alternative graph configurations. Each of these alternatives meets the functionality constraints of the enclosing compound object, and may share components with other configurations. Furthermore, the compound object is associated with a finite state machine (FSM) whose states are mapped to the alternative structures of the compound. The set of state transitions implemented by a given FSM models the configuration changes supported by the associated compound object. State transitions are triggered by events, which may originate from user input, the occurrence of particular data within the DSP, or an exception raised by a processing element.

Whenever a state transition (reconfiguration) is performed, part of the active processing graph needs to be removed and replaced with a new subgraph corresponding to the destination state. [SKB98] does not go into detail as to precisely how such a transition is effected, other than to indicate that reconfiguration actions are implemented as methods of the builder objects—hence the requirement that the builder object graph be retained after initial system instantiation. The authors do raise the question of how reconfiguration affects the ongoing behaviour of the overall DSP system: if a replacement subgraph starts up producing a different output value than its predecessor, it is liable to cause unwanted transient behaviour in the larger system. Ideally, the state of the old subgraph should be mapped onto its replacement such that its output does not change abruptly when the
transition takes place. Devising a generic mechanism to map state in this way is an open research problem; [KS99] explores a number of possible strategies but does not offer any globally applicable solutions.

This work is interesting because it tackles a problem domain that shares many characteristics with multimedia applications. DSP systems typically run with real-time constraints, such that it is not feasible to suspend the operation of all or part of the application—by placing it in a quiescent state, for example—in order to carry out a reconfiguration. Similarly, application integrity is as much a function of maintaining a valid real-time data flow as it is of ensuring a structurally consistent configuration of processing elements. The problem of unwanted, possible inconsistent, dynamic transient behaviour caused by mismatched state before and after a reconfiguration raises many of the same issues addressed by this thesis, specifically the requirement for reconfigurations that maintain consistency with respect to structure, data and temporal behaviour.

2.2.3 Distributed systems reconfiguration at Imperial College

The Distributed Systems Engineering (DSE) group at London's Imperial College has contributed immensely to the state of the art in distributed systems composition, reconfiguration and management technology over the past two decades. A common theme running through the DSE group's configuration research is the use of a language-based approach to building distributed applications. This methodology "combines the simplicity and safety of a language approach with the flexibility of an operating systems approach." [MKS89]. Moreover, the technique facilitates configuration, modularity and reuse of software modules—programming in the small—from the language used to configure these pre-written modules into a distributed program—programming in the large [DK76]. The benefits of this separation are confirmed by a large body of past research at Imperial College and elsewhere; for example, the HPC system [LF95], on-line maintenance of complex software systems [Sch95, HW96] and composition techniques for large-scale manufacturing automation applications [Lim96].

The various configuration programming and management systems developed at Imperial College have invariably made use of a specialised configuration language to describe the arrangement of self-contained software components into complete distributed programs. A similar approach has been adopted by numerous other research projects. For example, Durre [BWD+93], POLYLITH [Pur94] and STILE [RS94] deal with general distributed applications while Pinto and Liningten [PL94] and DAMSEL [PS95] specifically address the specification of multimedia systems. The specification of a distributed system provided by the language can be used both as a description of system structure [KM96] and to instantiate the actual running system [MDK94, MDK94a]. The configuration language itself has undergone considerable evolutionary from the Conic [MKS89] and REX [KMS92] systems through to the current version known as Darwin [MDK92, MDE95]. In more recent work
the language has been augmented by a graphical syntax and tools which allow interactive manipulation of system structures [FS96, Fos97a].

As with the configuration language, the distributed systems programming model supported by the different development environments has evolved over time. However, its basic features have remained essentially the same. Applications are constructed from groups of components, which are strictly typed in terms of the services they provide to other components, and those they require from other components [PC96]. Components can be hierarchically composed, with the behaviour of a composite component being determined by the functionality of its contained primitive components. Components make use of each others' services by binding a required service of one component to a provided service of another. Primitive ("atomic") components are written in an implementation language, usually C++. It is important to note that, in the model adopted by the DSE group, composite components have no runtime behaviour of their own, and exist only in the system description—the running program is made up entirely of atomic components. That is, the instantiation of a composite component involves the creation and interconnection of the atomic computational components at the bottom layer of the component hierarchy. No runtime objects corresponding to composite components are created.

Requirements for dynamic change

In [KM90] Kramer and Magee present a set of objectives for a dynamic change management system. The principles embodied by these objectives have been carried through into much of their subsequent work and have been influential in the system design presented by this thesis.

- Changes should be specified in terms of system structure. Attempting to perform changes at the level of individual atomic components is impractical due to the level of detail involved, whereas the effects of change at the more loosely-coupled composite component level can be clearly understood.

- Change specifications should be declarative. That is, the specification of the required changes should be separate from the details of how the change is to be carried out.

- Change specifications should be independent of the algorithms, protocols and states of the application. In order to provide generic configuration management, there must be no dependencies between the application and the configuration management system.

- Changes should leave the system in a consistent state. A consistent state is informally defined as one from which the application may continue normal processing, rather than progressing towards an error state. The application may of course pass through inconsistent states during the progress of a configuration change.
Changes should minimise the disruption to the application system. Many systems cannot be shut down or disrupted for extended periods, so changes should be executed promptly and only interfere with those parts of the system actually affected by the change.

The Darwin configuration language

Darwin is a declarative structuring language that allows distributed programs to be constructed in terms of parameterised, communicating component instances. A typical distributed program specified using Darwin consists of multiple instances of a limited set of component types. Darwin describes component in terms of their required and provided services, which are viewed as typed communication objects. In contrast to many other distributed systems environments (CORBA, for example), Darwin components make their service requirements explicit to the configuration management system. This allows for third-party bindings between required and provided services, where the binding is made by an external party and the components involved do not play an active role in the binding [CDF95]. If only first-party bindings are allowed, configuration information is often hidden within components, hindering reuse.

Darwin's principal function is to construct composite components, from both atomic computational elements and other composite components. A complete Darwin program is in fact simply a hierarchically structured composite component that declares some application-specific structure [MDK94a]. Composite components are created by declaring the component instances that they encapsulate and the bindings between the service interfaces of those components. Binding operations also declare the relationship between external composite component services and the services of the internal sub-components.

Dynamic configuration changes

Later versions of Darwin have incorporated two extensions allowing the declaration of programs with dynamic structure—lazy instantiation and direct dynamic instantiation [MDK94b, MDK94a]. Lazy instantiation was introduced by [MDK92] and is intended to cater for situation where the required number of components is not known until execution time. To quote Magee, Duly and Kramer in [MDK94a]: "The advantage of this technique of specifying dynamic structure is that the configuration description is a precise specification which describes the potential structure at execution time. The components used in the structure need not be aware of whether they are being used in a statically or lastly elaborated structure."

The lazy instantiation technique used by Darwin has two main drawbacks. Firstly, it does not allow parameters to be passed to new instances; and secondly, it is generally only suitable for recursive structures. In order to provide greater generality in the program structures that can be built, the version of Darwin developed for Regis incorporates support for direct dynamic instantiation, at
execution time, of both atomic and composite components. A component wishing to dynamically create other components of a particular type must specify this need in its Darwin definition, as another required service declaration. The requirement is satisfied by binding to a system-provided object that acts as a "factory" for components of the specified type. In the Darwin code, the binding is in fact made to the type of the desired components, rather than to a particular instance.

Dynamically created components are effectively anonymous, so it is not possible for Darwin to declare bindings to services provided by such components. However, bindings to required services of dynamically instantiated components are acceptable, since these can all be met by a single service provision in the static part of the application. Dynamically created components can provide services, but these can only be accessed by passing service references amongst components to make the bindings dynamically. Such bindings do not appear in the configuration specification, and thus serve to obscure program structure. They are however necessary for very dynamic or irregular structures.

Interactive configuration management

More recent research conducted in the DSE group involves environments for interactive management of distributed configurations [PS96, Fos97b, Fos97a]. A crucial difference between this work and previous systems is that the interactive environment—called ICON—deals with the configuration of actual running systems, rather than simply instantiating a system from a Darwin description. A problem with the Darwin approach is that the hierarchical system specification is effectively "flattened" when the application is instantiated at runtime, and knowledge of the system structure is lost. Interactive tools allow a configuration to be monitored and modified without reference to the original Darwin specification. The ICON model also includes facilities for maintaining and evolving persistent representations of running configurations. The environment can detect failed components within a running application and re-instantiate these from a persistent representation of the system.

The ICON system makes extensive use of a domain service [CDF+94]. Domains provide a means of grouping objects and specifying policies that apply to the members of a domain. Component interfaces are registered in management domains, which are used to group together interfaces that a manager is allowed to bind. Management domains are also used to, for example, group files representing new component types or physical nodes. Configurable component instances are registered in the domain hierarchy as configuration domains. These are similar to management domains in that they implement the same operations to, for instance, add and remove members of the domain. Configuration domains have additional capabilities specifically to support interactive configuration management and are implemented within the managed application. That is, a composite component is represented by a domain hierarchy that is created and managed within and by the component itself. The major benefit of this domain-based representation is that it allows managers to view and modify the configuration state of the application without referring to the Darwin code that was
initially used to elaborate the program.

The ICON configuration tools can be used to modify the structure of an application to such an extent that it no longer bears any relation to the Darwin description used to generate the initial structure. The configuration manager can generate Darwin code describing the actual configuration at any point in time, effectively creating a new composite component template that can be instantiated later. Note, however, that this facility can only produce code that would statically re-instantiate the system exactly as it was when the snapshot was taken: it is not, in general, possible to recreate the Darwin code relating to loops and conditionals.

Discussion

Although maintenance of consistent application state is stated as an important requirement for dynamic change management in [KM90], later work does not always specify how this goal is to be achieved. In the case of a system such as Regis, where designers are encouraged to express all dynamic configuration changes in the initial Darwin specification, this is perhaps not a significant issue. Even with Regis though, computational components have the ability to initiate configuration changes that are not explicitly declared in the Darwin code. The problem is potentially even worse in the case of ICON, which facilitates interactive, evolutionary change, perhaps initiated by managers with no knowledge of the original system specification. Magee, Dulay and Kramer recognise the problem in [MDK94b]: "This serves to obscure program structure, but is necessary for very dynamic and irregular structures. Where possible, we prefer to describe structure explicitly."

The domain system allows the specification of access rules which specify "a relationship between managers (in a subject domain) and managed objects (in a target domain) in terms of the management operations permitted on objects of a specific type. The access rule may also define constraints on these operations." [CDF+94]. Properly designed, such access rules could restrict the range of configuration changes that could be carried out on an application as a whole. However, they do not appear to provide a general way of specifying constraints or invariants on the relationships between components, particularly at the composite component and application level. Since we envisage that distributed multimedia systems will be managed interactively in a similar manner to ICON programs, and that they will want to maintain application-specific constraints independently of individual component behaviours, it is important that these can be specified in a general way and validated when configuration changes occur.

On a related note, complex configuration changes will involve many operations, including creation and deletion of components, and modification of inter-component bindings. In general, it will necessary for all of these operations to complete successfully in order to achieve a consistent state. Thus, we have the notion of an atomic configuration change, where either all of the operations complete, or none of them do, and no permanent changes are made to the system until a consistent state can be guaranteed. The Imperial College framework does not provide explicit support for
atomic configuration changes. The persistent configuration mechanism of ICON can be used to
roll the system back to a previous state, but the manager must remember to take a snapshot of the
affected components immediately before starting the reconfiguration, in case an operation should fail.

I believe that the problems discussed above stem, at least in part, from a lack of programmed run-
time behaviour in composite components. Prior to ICON, the elaboration of a Darwin program
resulted in a flattened structure containing only atomic components. Atomic components can, in
theory at least, perform arbitrary configuration actions, but this requires supposedly generic atomic
components to have detailed knowledge of high-level application structures. The limited configura-
tion actions available to composite components—lazy and direct dynamic instantiation—are
actually carried out by system-provided computational components. Composite components do
have a presence in the domain system, which allow new components to be instantiated and generic
configuration actions to be performed, such as the re-binding of services. There is, however, no
support for type-specific actions; a manager cannot, for instance, invoke an operation on the bank-
ing application described in [Fos97a]. Instead, the new component must be explicitly created in the
appropriate domain and all of its bindings configured by hand.

2.2.4 The SIRAC project and Olan

The ongoing "Systèmes Informatiques Répartis pour Applications Coopératives" (SIRAC) project
at INRIA Rhône-Alpes has also tended to follow a language-based approach to distributed system
reconfiguration. The Olan configuration language [BR96] also views applications as hierarchies
of connected components. Olan extends the original concepts embodied by Darwin and includes
many of the ideas that have also appeared in the Regis system. As well as the usual required and
provided services, Olan components may define broadcast events called notifications, and corre-
ponding reactions that are triggered when particular notifications arrive. Components may also
export some of their internal state in the form of shared variables or attributes.

Like Regis, Olan supports a variety of connector types [BBRVD98]. In theory the behaviour of these
connectors can be specified in detail using, for example, a set of constraints to describe QoS require-
ments, although the implementation of this functionality is not described in detail. In the area of
dynamic configuration, Olan has approximately the same capabilities as Regis: components may
be defined statically at compile time, or created at run-time using the lazy instantiation technique
found in Darwin. In addition, Olan supports collections of components, which are very similar to
HPC's interface bundles [LF85]. All of the members of a collection must be the same type, and the
size of the collection is restricted by both upper and lower bounds.

2 Distributed Systems for Cooperative Applications.
Dynamic reconfiguration of agent-based applications

More recent SIRAC work extends the reconfiguration capabilities of the Olan-based platform. In [PBR99] de Palma describes a dynamic reconfiguration service for an agent-based distributed systems middleware, utilising an Architecture Description Language (ADL) derived from Olan. The term "agent" in this work refers to the reactive software objects that are the basic building blocks of applications. Inter-agent communication is by typed events; agents react to the receipt of an event by executing a piece of code that can modify the agent's state and send further events to other agents. The communication system is reliable and enforces a causal ordering [Lam78, CDR94] property on event deliveries. Furthermore, agent state is persistent and agent reactions are atomic.

The goal of the dynamic reconfiguration mechanism is to "[maintain] application consistency while minimizing the impact on the running application." Consistency is defined by the relationship between agents' states and is held to be maintained whenever the reconfigured application is able to resume computation from the same global application state that existed before the reconfiguration. De Palma identifies a number of issues that must be addressed to meet this goal: naming preservation, state preservation and the status of the communication channel. Reconfigurations are modelled by modifications to the ADL description of the application; this approach has the significant advantage that changes can be validated before they are applied to the running agents. Thus, only globally consistent application structures should exist after a reconfiguration. It is then the job of the reconfiguration mechanism to ensure that agent reactions executed during the transition between two globally consistent states do not violate application consistency.

Agents may be in one of three execution states: their normal active state; a passive state where the agent can react to but not send events; and a frozen state where the agent can neither send nor receive events. The reconfiguration algorithm moves agents between states in order to maintain consistency during reconfiguration. In most cases, the target agent of a given reconfiguration action must be frozen before it is reconfigured, and a set of agents "upstream" from the target will need to be passivated to meet the preconditions for the freezing operation.

De Palma compares his algorithm to the reconfiguration mechanism used by Conic [MKS89] and demonstrates that his approach often requires far fewer agents to be passivated or frozen to safely perform a given reconfiguration; that is, there is less overall disruption to the application. However, the agent-based model is far less general than the Conic model and can take advantage of the causal ordering property to assist in maintaining consistency. These limitations aside, the architecture is significant in that it explicitly aims to reduce the impact of reconfigurations on the larger application.
2.2.5 Discussion

This section has presented a number of systems from the literature, that support dynamic runtime reconfiguration of distributed systems. The emphasis in this work is on general distributed systems rather than the specific needs of distributed multimedia.

With a single exception, the platforms discussed here rely on the notion of passivating part of the distributed system during reconfiguration to ensure that an application remains globally consistent before and after the configuration change. Moreover, the SIRAC work in particular supports transactional reconfigurations, allowing complex changes to be made atomically and rolled back if any part of the reconfiguration fails. The exception is the self-adaptive DSP platform examined in Section 2.2.2, which addresses the problems inherent in reconfiguring a system with both continuous data flow and real-time processing constraints—properties that are also characteristic of multimedia systems. This platform is also distinguished by its use of an explicit runtime model of the DSP system, which is used to drive the reconfiguration process. The model retains the full compound structure of the original system design and allows each compound object to implement its own type-specific reconfiguration behaviour. That is, the deployment of reconfiguration functionality is not centralised within the application.

In contrast, the language-based approach of Imperial College and SIRAC typically gather all reconfiguration functionality together in a single place—the high-level language description of the application. Darwin, in particular, requires that the system specification enumerate all of the possible reconfigurations that the application may undergo during its lifetime. Such restrictions do serve to ensure that the application can only be reconfigured into known stable, consistent states, but imply firstly that the programmer is able to describe all expected reconfigurations before the system is deployed and secondly, that no unexpected reconfiguration requirements are discovered once the system is running—an unlikely situation for a long-lived, adaptive application. It is also worth noting again that composite objects in both Darwin and Olan do not have an explicit runtime presence once the application has been instantiated, and thus cannot offer runtime reconfiguration support to their sub-components. Reconfiguration must be driven either by the lowest level atomic components or from the very top level, through the application description.

2.3 Adaptive and reconfigurable multimedia systems

This section examines a number of important systems that attempt to address issues in the design and implementation of adaptive and reconfigurable multimedia systems. As with the previous sections, this is not intended to be a comprehensive survey of work in this area; rather, it seeks to

---

3More recent Imperial College work, notably ICON, does relax this restriction and allow application structure to be modified at runtime without reference to the original Darwin specification. However, by doing so many of the consistency and management advantages of the language-based approach are lost.
illuminate the main research issues and, in particular, to highlight those systems which have been influential in the development of the work to be presented in later chapters. The work discussed here typically builds on the basic multimedia framework concepts embodied by the systems covered in Section 2.1, and endeavours to extend these with explicit support for dynamic reconfiguration and adaptation of running applications, beyond the simple rate-control feedback mechanisms found in the other projects.

2.3.1 CINEMA

The Configurable INtEgrated Multimedia Architecture (CINEMA) [HR94, Hel94] was developed in the Institute of Parallel and Distributed High-Performance Systems (IPVR) at the University of Stuttgart. It aims to provide a set of abstractions for the dynamic configuration of distributed multimedia applications [RBH94].

A CINEMA application consists of one or more clients and a set of data flow graphs. A client is a software entity that uses CINEMA services to firstly define, then control and modify, its associated data flow graphs. The graphs describe the media data flows through networks of interconnected media processing components. This architecture appears superficially similar to the split-level approach of the VuSystem and its descendants, where a high-level script defines and controls a graph of media processing components. There is, however, a subtle yet fundamental difference: the components in a data flow graph created by a VuSystem application are the actual media processing objects, whereas a graph created by a CINEMA client is merely a representation or a model of the “real” components; that is, the objects in a CINEMA graph do not themselves process any media data. The systems described in the previous section demonstrate that the ability to reason about the structure of an application independently from its implementation is a powerful tool; the CINEMA architecture provides this ability and in addition allows a client to dynamically and consistently reconfigure a running application.

From a client’s point of view, a CINEMA component encapsulates one or more low-level hardware or software devices. The exact devices covered by a given component are specified when the component is instantiated. A component also exports several interfaces to the client: ports, where each port has an associated stream type used for validating connections; a type-specific component control interface; and a clock interface used to control the timing of media data units. Component types are defined using a custom “Component Programming Language” (CPL) based on C++. A CPL program defines the exported interfaces of the component type, and an action method—the media processing code for the component, that will be periodically run in a real-time thread. The CINEMA developers claim that the use of CPL assists in achieving configuration independence for components. However, no convincing evidence is provided for this statement: CPL appears to be little more than a set of C++ macros. It is true that component creation uses an indirect mechanism—the component class is specified as a text string rather than compiled into the client
code—that would permit component implementations to be loaded dynamically at runtime. This facility would ideally be coupled with a reflection mechanism allowing a client to dynamically query and invoke previously unknown methods on a component; it is not specified whether such functionality is available.

CINEMA supports composition of components to an arbitrary degree. A compound component has no action method; instead, it specifies the structure of its internal component graph and the mapping between ports on the encapsulated components and those on the enclosing composite shell. Similarly, a compound component defines its own component control interface—the control interfaces of the internal components are not exposed and may only be invoked via methods of the compound class.

Sessions and dynamic reconfiguration

CINEMA recognises that an architecture that only offers control over individual components, or even individual media streams, is insufficient. Many application scenarios require multiple streams to be grouped and handled together due to synchronisation or other relationships between the streams. For example, a video-conferencing application might require that sufficient resources are available to support simultaneous transport or audio and video streams, synchronised to within 50ms of one another. Sessions specify those parts of a data flow graph that firstly, must be established atomically—that is, in an all-or-nothing fashion—and, secondly, must meet and maintain a certain quality of service. Successful creation of a session results in the physical realisation of the logical structure described by a data flow graph. Processing of media data can only be started after a session has been established.

In [Bar96], Barth describes algorithms for assigning components to threads and for determining the sets of components affected by session creation and deletion operations. Dynamic reconfiguration of the active application is possible, by selectively adding or removing graph segments to or from sessions. CINEMA will compute the set of changes to the physical application structure implied by these operations and ensure that the entire change set is applied atomically. Quality of service issues and implementation are discussed in detail by [RDF96]. Unfortunately, the CINEMA literature does not offer any insight into how the temporal consistency of a data flow graph is maintained during reconfiguration. For example, consider a reconfiguration that deletes a consumer component and replaces it with another. Intuitively, the deletion should be performed first, so that the new component can be successfully connected. However, if the new component is not instantiated quickly enough, media data may be lost while neither sink component is connected. The CINEMA architecture does not attempt to address this issue.
2.3.2 QUASAR

The QUAlity Specification and Adaptive Resource management (QUASAR) project in the Department of Computer Science and Engineering at the Oregon Graduate Institute (OGI) is investigating issues in QoS specification for both adaptive and reservation-based resource management systems. The research has proceeded along two parallel tracks: QoS specification and software feedback mechanisms for adaptive multimedia systems. This section concerns itself solely with the second of these areas.

The QUASAR approach to adaptive software design is inspired by feedback control theory, a common technique in other engineering disciplines [GSPW98]. While feedback mechanisms have been used previously in computing applications—congestion control in the TCP protocol [Pos81, Jac88, JBB92] and synchronisation of continuous media streams [LG90, AH91, Sre92, RR93, JSN95], for instance—the QUASAR researchers believe that these existing designs have two problems [GSPW]. Firstly, they suffer from implicit assumptions about their operating environment. For example, the TCP protocol was designed to operate over wired networks with reasonably well-known characteristics; consequently, it performs poorly when run over wireless networks [YB94, MJS96]. Secondly, feedback controllers have typically been implemented in an ad-hoc fashion, tightly bound to a particular application and difficult to reuse or maintain. To solve these problems, QUASAR proposes an architecture for modular, dynamically reconfigurable and analysable feedback controllers, embodied by the SWiFT framework [GSPW98].

SWiFT controllers are composed from feedback components, software objects very similar in concept to the processing components of the VuSystem and most other multimedia programming systems (Figure 2.3). Feedback components read data from their input port(s), calculate an output value via an encapsulated transfer function, and pass the value to their output port. Monitors and actuators are special components that connect the controller to the controlled system; they have no input or output ports, respectively. Controllers may also be parameterised, analogous to fitting knobs and switches to a physical control system. This model closely resembles the way real-world electronic feedback circuits are built up from individual components. The SWiFT architecture also support hierarchical composition of feedback components, using special components known as feedback containers. A container defines a circuit of connections amongst its enclosed subcomponents, and maps any unconnected ports onto its own input and output ports.

Dynamic reconfiguration

The purpose of a feedback controller is to provide control inputs—via the actuator components—to the controlled system, thus managing its operation. The feedback loop is completed by status outputs from the controlled system, connected to the monitor components of the controller. SWiFT supports dynamic reconfiguration of a feedback system in three ways:
Figure 2.3: SWiFT feedback control model (from [GSPW98]).
1. The parameters of one or more feedback controller components can be altered.

2. The controller can be reset, essentially forcing it to converge onto a new operating point from a known initial state.

3. Components of the controller can be added, removed or replaced. This method is used in response to major changes in the operating environment of the controlled system—such as a mobile terminal moving from a wired to a wireless network—that violate the design assumptions of the controller.

Reconfiguration is controlled by user-specified predicates known as guards. These are simple upper and/or lower bounds on the input values, output value or internal state of the controller that initiate reconfiguration of the controller whenever the monitored values move outside the specified range. It is important to note that a reconfiguration of the controller can only affect the control inputs fed to the controlled system, not its internal structure and connectivity—although certain control inputs could conceivably instruct the controlled system to make such changes. This limits the range of runtime reconfigurations that can be performed to those explicitly anticipated by the system designer, for which there are suitable controller configurations available.

**Multimedia applications**

The QUASAR Internet audio/video player [WKC+97, ICPW97] utilises the feedback control architecture of SWIFT. The player dynamically adapts to changes in operating conditions and resource availability—CPU bandwidth, network bandwidth, latency and jitter—and supports multidimensional media scaling based on user-specified QoS requirements.

The player architecture consists of a number of server processes serving MPEG video and MPEG or μ-law audio streams across the Internet to one or more clients. A client may be receiving streams from multiple servers simultaneously. Media data is transferred using the Streaming Control Protocol (SCP) [CPW97], a UDP-based protocol optimised for streaming continuous media data. During playback, a client may instruct the server to change the quality of the stream, or adjust its timing parameters. Each MPEG video stream is stored at four different resolutions, and users may manually adjust their desired playback frame rate and image size.

Because the player application operates in a best-effort mode, it may drop video frames at any point between the server and client where resource shortages exist. The further along the processing pipeline a frame is dropped, the more resources have been wasted: for instance, a frame that is discarded immediately before display, because it arrived late, has already consumed resources in the server, network and client-side decoder. The player uses a feedback mechanism that attempts to adapt the resolution and rate of the video stream such that frames are only ever discarded by the server; this has the additional advantage that the server can leverage its knowledge of the MPEG
stream structure to intelligently choose which frames to drop, maximising the number of frames successfully decoded and displayed by the client. The input to the feedback controller is the rate at which frames are actually displayed by the client; that is, the end-to-end frame rate. This data is used to adapt the rate of the server to a level that fits within current resource constraints and the capabilities of the client. A second feedback loop monitors the change in end-to-end latency (jitter), and adjusts the amount of buffering used by the client.

2.3.3 ADAPT

The ADAPT project [FBC+98a, FBC+98b, BCD+97] is part of a large collection of ongoing work conducted by the Distributed Multimedia Research Group (DMRG) at the University of Lancaster. ADAPT is classified under the heading of Next Generation Middleware, which includes other notable projects such as SUMO [CBR94, CB95] and Retina [BEC99]. Members of the DMRG have been influential in developing the multimedia aspects of the ISO Reference Model for Open Distributed Processing (RM-ODP) [BS98]. A key premise of the ADAPT research is that future distributed applications must be able to adapt to changes in network QoS arising as a consequence of user mobility. The project is particularly concerned with middleware-layer support for the adaptation functionality required by multimedia applications. A variety of other projects take a similar approach in attempting to dynamically adapt to fluctuations in environmental conditions, although their resulting implementations are very different. For example, in [FGCB98] Fox, et al. describe a proxy-based solution while [CBS98] and [LN99] present systems based on benefit functions (see also [NKMC98]) and fuzzy QUASAR-style feedback control loops, respectively.

Middleware platforms such as CORBA typically take a "black box" approach where objects are accessed solely through their defined interfaces; all details of the underlying implementation are kept hidden. Fitzpatrick [FBC+98b] argues that this architecture is unsatisfactory for mobile middleware applications, which require access to certain aspects of the implementation in order to successfully adapt to changing conditions. ADAPT makes use of concepts from the areas of open implementation [KRB91] and reflection [Mae87] to address this need. The goal of open implementation is to open up key aspects of the underlying implementation to applications, in such a way that a clear distinction is maintained between the functionality supplied by a module (the base interface) and its implementation (the meta-interface). Reflection provides a means of achieving this goal. The meta-interface to a reflective system supports manipulation of a "causally connected self-representation of the underlying implementation." Maes defines a causally connected system as one where

"...the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect on the other." [Mae87]

A reflective system naturally supports inspection and adaptation of its underlying implementation.
Explicit and open bindings

In the existing CORBA programming model, bindings between objects are made implicitly, that is, the ORB automatically creates a communications path between interacting objects. ADAPT extends this model—in the process bringing it closer to the RM-ODP approach—to support bindings that are both explicit and open (Figure 2.4). An explicit binding is made by instantiating an object representing the end-to-end communications path between two interacting objects. Bindings may be created by third parties, as well as by the endpoints of the connection. In addition, a binding object exposes a control interface that can support reflective inspection and adaptation functionality.

Two styles of binding are available: operational bindings are the explicit form of traditional CORBA bindings, supporting request-reply invocations between the connected endpoints. Stream bindings support continuous media streams, where a stream consists of a number of unidirectional flows, each of some media type. Bindings are created by a set of binding factories, specialised objects whose function is to instantiate binding objects with particular semantics.

In order to support inspection and adaptation, bindings offer a meta-interface that gives access to a causally connected self representation of the binding implementation. As might be expected given the ubiquity of this structure in the preceding discussions, the self representation takes the form of an object graph modelling the end-to-end data flow between the binding endpoints. An object graph consists of processing objects and binding objects connected by local bindings; the non-local bindings explicitly represent the interactions between objects in the graph. Strict type-checking is enforced for all local bindings, to ensure at least a basic level of consistency in the structure of object graphs. Binding objects in the graph may themselves be open bindings, composed from other object graphs; the innermost level of the composition consists of primitive bindings with closed implementations. As with the G&T framework, it is unclear why bindings (cf. G&T’s connectors) must be distinguished so obviously from “processing” objects, when it can be argued that the commu-
nication function of a binding is just a specialised processing behaviour. Likewise, it is not clear whether the architecture allows non-binding processing objects to be composed in a similar fashion to bindings.

**Reconfiguration of object graphs**

Object graph configurations may be modified by the addition or removal of processing/binding nodes; local bindings should also be changed appropriately to maintain a properly connected graph structure. To defend against arbitrary reconfigurations damaging the functional characteristics of a binding, changes must be requested via the meta-interface of the enclosing component, thus allowing it to reject inconsistent configuration changes. However, often a reconfiguration will consist of a sequence of actions that will collectively leave the object graph in a perfectly consistent state, but when considered individually, would be rejected by the meta-interface. For example, replacing one object with another by first deleting the old object, then inserting the new one, will leave part of the object graph briefly disconnected between the two steps; thus, the meta-interface may not allow the initial deletion to take place. This problem is attacked by defining a set of compound graph operations, encapsulating common sequences of reconfiguration actions such as object replacement.

While this approach may solve the immediate problem of replacing a single object, it is not a general solution for all complex changes. Consider a reconfiguration that requires the replacement of two separate objects; further, assume that the first replacement operation succeeds, but that the second fails because some required resource is not available. The object graph is now in an inconsistent state, which may lead to incorrect application behaviour, but this condition has not been detected by the meta-interface. A more general mechanism would permit an arbitrary sequence of operations to be grouped together as a single reconfiguration and applied atomically, such that if any operation failed the state of the binding object would remain unchanged.

**2.3.4 ATLANTIC**

The activities performed in a modern digital television (DTV) studio are broadly the same as those carried out in its analogue counterparts of previous years; indeed, the interconnection and control of studio equipment remains firmly grounded in the tried and proven techniques of earlier generations. The use of digital multimedia systems is generally restricted to offline, non-real-time editing and the production of complex computer graphics sequences. Mixing and switching of live, recorded and computer-generated video signals for broadcast is still largely a manual task, with human controllers pushing sliders or flipping switches. Additionally, the processing and switching equipment used in a studio is most often special purpose, proprietary hardware communicating

---

4In fact, [FBC+98b] seems to indicate that the ADAPT framework is implemented in this way, with bindings being a subclass of the generic component class.
over dedicated low-level links. However, as the television industry moves towards the time when analogue broadcasts will no longer be available, there is a small but growing movement—in the broadcasting research community if not yet amongst production staff—away from this low-level viewpoint towards a more open, middleware-style approach. This shift will bring DTV systems at least partly within the sphere of distributed systems engineering research. For example, video compression techniques are increasingly being applied to every part of the programme production and distribution chain [BDE97], including:

- Programme origination
- Programme post-production (editing)
- Contributions from external sources (carried over satellite or landline networks)
- Transmission to regional centres (who may insert their own local programming)
- Distribution to consumers/subscribers (via cable, satellite or conventional broadcast)
- Archiving

The television community has agreed on the use of MPEG-2 [ISO93, ISO96] coding for programme distribution and it is likely that this will also become the de-facto coding standard used within studios for production, post-production and archiving purposes. If we take it as given that compressed MPEG-2 streams are to be used throughout the programme production and distribution process then clearly these streams must undergo manipulations formerly carried out on uncompressed streams:

- Transitions between two (or more) streams, e.g. cuts, cross-fades or other more complex operations.
- Insertion of captions, logos and other graphics.
- Frame-accurate editing.
- Format conversions, e.g. changing the picture size or bit-rate (this is often referred to as transcoding, especially when dealing with coded stream formats).

One of the most important such manipulations is the ability to switch between two compressed streams [BBH97, KW97]. Switching is required, for example, in post-production editing, "continuity switching" between different studios or external sources, and in "opt-out" switching to local programming or commercials. The switching operation is relatively trivial for analogue or uncompressed digital streams and is generally performed during the "blanking interval" between frames; that is, the viewer will see a complete frame from the first stream followed by a complete frame from
the second stream, with no unwanted artifacts caused by the switching. The *switch point* (the point in time where the switch occurs) is easily determined since the two streams are tightly synchronised and each frame occupies precisely the same number of bits in the stream.

Unfortunately the switching operation is not quite so straightforward in the compressed domain. Encoded pictures generally occupy different amounts of time and/or bits, making it more difficult to locate and synchronise the switch point. Furthermore, the MPEG-2 encoding uses temporal prediction techniques; in particular it may be impossible to decode a given frame until a certain *future* frame has been decoded—which becomes inconvenient if the required frame lies *after* the switch point and is thus lost when the switch occurs.

SMPTE\(^5\) has proposed a standard for splicing together MPEG-2 streams in the transport format commonly used by broadcasters. Splicing can only occur at specified *splice points* encoded into the source streams, with further restrictions on the splice point that is actually used. This technique is unsuitable for applications requiring frame-accurate switching, most notably editing.

An alternative approach is to decode the incoming streams, perform the switch in the uncompressed domain, then recode the resulting output stream. This method has the advantage of imposing no restrictions on the location of the switch point, and a wider variety of transitions are possible using existing processing equipment. The drawback is that the picture of the recoded stream will be visibly degraded, since the coder will almost certainly choose different encoding parameters to either of the original compressed streams. Multiple re-encodings of a stream—not an unlikely occurrence over its lifetime—quickly result in unacceptable picture quality.

The ATLANTIC project at BBC Research and Development has developed a novel MPEG-2 coding architecture that can perform multiple cascaded re-codings of a stream with negligible degradation in picture quality [TW97, BDK97, KW97]. The system also supports frame-accurate switching of compressed streams. ATLANTIC's technology is based around the idea of the *info-bus*, a separate data stream—carried in parallel with a decoded stream—that contains all of the relevant information on how the stream was encoded. The info-bus signal is generated by an enhanced decoder when it decompresses an incoming MPEG-2 stream. It can then be used by a cooperating downstream encoder to recode the stream in exactly the same way as the original. The only mismatches between the original and recoded streams are caused by asymmetries between the Discrete Cosine Transform DCT and inverse DCT transforms and are negligible in practice [KW97, Th99]. It should be pointed out that the info-bus only carries information that was present in the original MPEG-2 stream, so that the downstream re-coder will effectively make the same coding decisions as the original encoder, but without the need to recompute all of the coding parameters.

The ATLANTIC switch performs the actual switch (or other transition) in the decoded domain, but uses the info-bus signals from both incoming streams to almost eliminate degradation in the

---

\(^5\)SMPTE is the Society of Motion Picture and Television Engineers, an important professional association and standards-setting body in the film and television industries.
switched output stream (Figure 2.5). In the steady state, before or after the switch point, either stream A or stream B is selected as the output of the switch. The chosen stream is decoded and recoded almost transparently, although some amount of additional latency will be introduced into the signal path by the switch components. When the switch point arrives, the input of the encoder is switched to the other incoming stream; this transition can occur on an arbitrary frame boundary. However, info-bus data from either stream is not directly usable by the coder for frames on either side of the switch point, because it will refer to frames on the other side of the switch point that the encoder has not seen. The switch is able to modify the contents of both info-bus channels in order to provide the coder with a consistent view of the incoming stream:

- The picture type (I, P or B) of a frame may be changed. For example, a P-frame just after the switch point can be recoded as an I-frame to remove the reference to an unseen frame from the second stream.

- Motion vectors and motion prediction modes may need to be modified or recalculated to account for changes in picture type or to eliminate predictions across the switch point.

- Quantisation parameters will generally need to be changed to prevent downstream buffer under- or overflow around the switch point. It may take some time for the encoder's quantisation settings to converge to those of the second stream; this interval is known as the recovery period. The recovery period can be reduced using a technique known as temporal masking, which takes advantage of the human eye's inability to distinguish even large amounts of noise in the video signal around a scene change—picture quality can thus be reduced significantly around the switch point without any observable degradation. The reduced bit rate of lower quality frames allows a shorter recovery period to be achieved.
More complex transitions such as cross-fades may be used instead of the simple switch described above. However, when the input to the encoder does not correspond directly to a frame from either of the input streams the info-bus signal becomes invalid and the frame must be re-encoded from scratch.

2.3.5 Discussion

This section has evaluated a number of multimedia programming frameworks that provide explicit support for runtime adaptation or reconfiguration of their applications. There are clearly a wide range of possible approaches to this area, from the relatively simple and inflexible feedback controllers of QUASAR through to the fully open and extensible architecture of ADAPT. What is also clear is that none of these systems provides a complete solution to the problem.

As with the majority of the non-adaptive platforms discussed in Section 2.1, all of the frameworks in this section adopt the familiar component-based architecture. Interestingly, in the case of QUASAR this architecture is applied with respect to the adaptation control mechanism rather than the media processing components themselves. QUASAR’s reconfiguration mechanism is limited by the fact that it can only modify the parameterisation of an existing VuSystem-style media pipeline; there is no support for changing the structural configuration of the system, to cope with adaptation outside the range of the extant components.

The split-level CINEMA architecture at first appears similar to the VuSystem approach. However, the crucial difference is that where VuSystem applications are constructed in a completely ad-hoc fashion using a scripting language, the upper layer of a CINEMA application is a object-oriented model of the underlying media processing structure. The model permits analysis of changes to the application structure before they are applied to the “live” components, in particular the atomic allocation of resources needed to support the QoS requirements of the new structure. Unfortunately, the model does not extend to the characteristics of the communication between distributed components. In contrast, ADAPT makes certain aspects of communication explicit through the use of open bindings. The principled and elegant design of the ADAPT architecture is let down, however, by the lack of support for complex adaptations involving multiple bindings and often causing transient inconsistency in the application structure. Bindings tend to be over-zealous in defending their own internal consistency, frustrating the higher level goals of their parent bindings and the overall application.

The BBC ATLANTIC system is perhaps something of a special case with respect to the other systems covered by this section: it supports only a single, highly specialised application and requires an expensive, dedicated hardware platform in order to do so. It was included firstly to illustrate some real-world application requirements and, more importantly, to demonstrate the need for smooth—temporally as well as structurally consistent—reconfigurations and precise timing control during
the often transiently inconsistent transition between two consistent configurations. Neither of these requirements are directly addressed by any of the other systems covered in this section.

2.4 Summary

This chapter has presented an overview of research in three related areas, concentrating on work that has been the most influential on the ideas developed by this thesis. Section 2.1 investigated distributed multimedia programming frameworks, Section 2.2 discussed reconfiguration mechanisms for general distributed systems and Section 2.3 brought these two threads together with an exploration of multimedia programming systems supporting online adaptation and reconfiguration.

Component-based systems appear to have become almost de rigueur in current distributed multimedia research, indeed, this is probably the case in the distributed systems field generally. However, as always the devil is in the details: there are clearly a multitude of ways of implementing a nominally component-based system. The Lancaster ADAPT architecture and the closely related continuous-media abstractions of the RM-ODP standard are particularly noteworthy: none of the other multimedia platforms surveyed here provide quite the same level of openness and extensibility that is offered by the open binding mechanism. Of course, this is not to say that RM-ODP provides a complete solution to all adaptive multimedia problems.

The remainder of this thesis describes a distributed multimedia programming architecture that attempts to address some of the deficiencies of other work in this field, as noted above. In particular, it proposes techniques for modelling the reconfiguration behaviour of component-based applications, and mechanisms to manage temporal as well as structural consistency during reconfigurations.
CHAPTER 3

REQUIREMENTS STUDIES

This chapter explores the requirements imposed by distributed continuous-media applications, with respect to a middleware software layer that provides programming and runtime support to applications and maps their high-level activities onto the low-level operating system and network services of a distributed computer system. The demands made on the underlying operating systems and networks by the middleware layer are also examined.

Distributed continuous-media applications have additional requirements beyond those of more traditional distributed systems. In particular, these applications manipulate streams of media data with soft real-time characteristics; the steady-state condition of a multimedia system is typically one in which media streams are flowing between system components. In contrast, the interactions between components in a conventional non-multimedia distributed system consist of discrete request-reply dialogues, messages or events. It follows that multimedia systems will require a different programming model to their conventional counterparts, with new system support mechanisms needed to fully realise the unique features of these applications.

The remainder of this chapter is structured as follows:

- Section 3.1 introduces two application scenarios drawn from the domains addressed by this research. The first of these is a mobile surveillance system based on a real-world application developed for the emergency services, while the second is a programme editing and production system for a digital television studio. This second application is rather more ambitious in scope and has yet to be fully implemented, although many of the underlying technologies are in production use. Both applications are described in detail and a set of high-level requirements presented for each one.

- Section 3.2 presents a list of fundamental properties shared by most distributed multimedia applications, developed from the application scenarios in the previous section and the systems...
reviewed in Chapter 2. These properties will be used to motivate the framework design and system support requirements presented in the following sections.

- Section 3.3 describes a comprehensive set of requirements for distributed, real-time, continuous-media application support. This encompasses both the design requirements for a distributed multimedia programming framework and the system support requirements imposed on operating systems and networks by these applications. The requirements were derived through analysis of existing programming frameworks, applications and system support layers—including the case studies from the previous section—and from discussions with professionals building and using distributed multimedia systems in the “real world”.

- Section 3.4 extends the previous section and focuses on requirements specific to dynamic runtime reconfiguration of multimedia systems. These requirements will provide the motivation for the reconfiguration mechanisms presented in Chapters 5 and 6.

3.1 Application Case Studies

In this section I will introduce the two distributed multimedia application case studies. These studies will be developed further in subsequent chapters as the details of the DJINN framework and my approach to reconfiguration are presented. For now, a relatively high-level discussion of these applications will serve to motivate and justify the set of requirements to be developed later in this chapter. The applications used in the case studies were chosen primarily because they exemplify most of the issues addressed by this research; they are typical of the application domains that I aim to support. The first application—a remote, mobile surveillance system for the emergency services—is well defined and relatively small in scope. It has been implemented as a demonstration system by a group at Lancaster University [YDFB98, DMCB98, DFFS98]. In contrast, the second case study deals with a more hypothetical application domain, at least from the perspective of middleware support: digital television production. While digital television studios clearly do exist and provide satisfactory performance to end users (the television-viewing public), the equipment and techniques used by these facilities are still deeply influenced by traditional analogue technologies. The second case study describes a more open and flexible approach to this application domain, albeit one that has yet to be put into practice. It is based primarily on my discussions [Tud99] with Phillip Tudor from the British Broadcasting Corporation (BBC) Research and Development division. The BBC is acknowledged as a world leader in the development of digital television. Most of the underlying technology presented in this case study has been implemented at least to the demonstrator stage, although the parts related to my research remain hypothetical at present.
3.1.1 Case Study 1: Remote Surveillance

This application scenario is based on that described by Yeadon, et. al. in [YDFB98]. These researchers at Lancaster University are developing systems to provide mobile multimedia support and applications for the emergency services. Their example is set in a bank; however, the scenario is equally applicable to any large security-conscious site such as a factory, a political convention or a research centre. Figure 3.1 shows the main components of the system. The site is equipped with fixed and mobile surveillance cameras, the latter carried by on-site security personnel. The video streams from the fixed and mobile cameras are delivered to a central server over 100Mbit/s fixed Ethernet [MB76] and 4Mbit/s wireless WaveLAN [Tuc93] networks respectively. Security personnel can monitor the live video via either conventional workstations on the Ethernet or mobile terminals using the WaveLAN. Mobile users who stray outside the WaveLAN coverage area may still be able to send and receive video—with significantly reduced quality—over a 9.6Kbit/s GSM [Rah93] cellular link to their mobile terminals.

In the event of a major incident—such as a factory fire or bank robbery—when the emergency services are called, the surveillance video streams will be routed to the police, fire brigade or ambulance control room over a high-speed wired link. An appropriate subset of the streams can then be forwarded to emergency units en route to the scene, again using GSM or a dedicated packet radio network. Once on the scene, emergency services personnel should be able to receive higher-quality video from the WaveLAN at the incident site. If audio streams are also available, these will be handled in the same way.

The three networks used by this system (Fast Ethernet, WaveLAN and GSM) have vastly different characteristics. Most significantly, there is a two orders of magnitude change in bandwidth between the Ethernet and WaveLAN networks and a similar disparity between the WaveLAN and GSM networks. This huge range means that different video encoding techniques are required for each network. For instance, the Ethernet can easily support a number of high-quality MPEG-1 [ISO93] streams at rates in excess of 15 frames/s. The WaveLAN also offers sufficient bandwidth to carry an MPEG stream, but it will probably be necessary to reduce the frame size and/or rate if multiple streams are being carried. At the bottom end of the scale, a very low bit-rate encoding such as H.263 [ITU96, CEGK98] is appropriate for the extremely limited bandwidth of the GSM network. Note that since GSM is a point-to-point technology, and the mobile handsets can only display a single video stream, it will never be necessary to carry multiple streams over a single GSM link.

Some key application-level requirements of the surveillance application, from its users' point of view, are listed below. These are adapted from a similar list in Yeadon, et. al. [YDFB98].

- **Heterogeneity.** The system must tie together systems owned and run by several different organisations (the secure site, the emergency services, network providers, etc) and allow these to cooperate within well-defined boundaries.
- **Real-time.** Emergency services have stringent real-time constraints for multimedia data [DFFS98]. The system must provide and meet guarantees on the delivery of audio and video streams. In addition, any required synchronisation relationships between streams must be maintained.

- **Configurability.** The remote surveillance system must be equipped to manage a constantly changing mix of local and remote users viewing different streams. Switching between networks should be performed seamlessly, that is, no stream data should be lost or corrupted during the switch from one transport to another.

- **Resilience.** The application must be robust in the face of failures, transient or otherwise. In particular, unsuccessful reconfiguration attempts should not be allowed to adversely affect the overall integrity of the application.

- **Availability.** A system such as this will likely need to be running 24 hours per day, seven days per week. It will generally be impractical or even dangerous to shut the application down completely for maintenance or upgrading. Therefore, the system should support online "evolution" of its structure and capabilities without disruption to its normal operation.

- **Security.** Ensuring the authenticity of, and restricting access to, stream data are clearly of paramount importance to this application.

Note that this work does not address the question of security, other than to note that it is an important issue. Also, fault-tolerance aspects of the resilience and availability requirements are not covered by this research.
Real-time continuous media data predominates in the surveillance application. An important consequence of this predominance is that traditional distributed systems engineering techniques for addressing the above requirements are not necessarily applicable in this scenario. Alternative techniques must be developed that explicitly take account of the application’s time sensitive nature.

### 3.1.2 Case Study 2: Digital Television Production

My second case study concerns a hypothetical digital television editing system based on the ATLANTIC MPEG-2 switching and coding technology described in Chapter 2. A prototype of this system has been built and field-tested [Tud99], although without any of the extensions proposed here. The ATLANTIC editing system is designed for a post-production environment where finished programmes are assembled from pre-recorded audio and video clips stored on a file server. From the producer or journalist’s point of view, the major functional requirements of this application are [BDR97]:

- Frame-accurate edit transitions, with independent audio and video transition times.
- A wide variety of transition styles, from simple cuts and fades up to complex effects.
- Frequent transitions possible.
- Editing functionality similar to that offered by current motion-JPEG based non-linear editing software.
- MPEG-specific details not exposed to the user.
- Scalable to a large number of simultaneous users.

Figure 3.2 shows the main components of the ATLANTIC editing system:

**Format converter.** This de-multiplexes incoming MPEG-2 streams into separate video, audio and data clips that are stored—still in their compressed form—on the main server.

**Browse track generator.** To provide full random access to the stored clips, and to support editing on low-cost hardware, a *browse track* is generated for each video clip and stored on a separate browse server. The browse track is generally an I-frame-only stream with reduced spatial resolution and picture quality.

**Editing workstation.** Producers and journalists use specialised editing software running on a conventional workstation to compose a programme from the stored browse tracks. The output of the editing software is an *edit decision list* (EDL) specifying which portions of which clips make up the finished programme, what order they should be played in and the transitions (cuts, cross-fades, etc.) between clips.
Figure 3.2: The ATLANTIC editing facility.

**Edit conforming switch.** This is a modified version of the ATLANTIC switch described in [KW97] which reads the stored streams from the main server and performs the transitions specified in the edit decision list. Source streams are routed alternately to the two decoding inputs of the switch; the EDL identifies the switch point and any additional processing that is required to achieve the desired transition. The recoded output is multiplexed together with a suitable audio track and stored on the programme server for later broadcast.

Using MPEG streams throughout the editing process has the major benefit that the compressed streams provide subjectively equivalent picture quality at a significantly lower bit-rate than the uncompressed streams typically used in studios. The ATLANTIC switching technology allows the editing to be performed without the risk of degraded picture quality after repeated recoding. Thus, the cost of servers and network infrastructure can be reduced considerably. Such economic considerations are important to the BBC, which is endeavouring to expand the range of channels and services it offers within the fixed income of the licence fee paid by television owners in the United Kingdom. The goal of the organisation is to reduce the cost per channel through the use of cheaper generic equipment rather than the expensive proprietary systems currently found in studios. A secondary goal is to make systems such as the editing application available to a wider range of users distributed throughout the BBC, in contrast to the present situation where a high-quality analogue editing suite, for example, is a limited resource that must be booked on an hourly basis.

The existing prototype editing system does not yet meet all the needs of programme producers. Their typical practice is to assemble, on videotape, a high-quality "rough cut" of the finished programme—containing somewhat more material than will be included in the final cut—generated

---

1These will generally be SMPTE "Recommendation 601" streams. A Rec. 601 stream is essentially a raw digitised video signal carried over dedicated cables.
from browse tracks using ATLANTIC or an existing motion-JPEG based system. Final editing is done on the uncompressed broadcast-quality material using multiple tapes in an editing suite. Tools have not yet been developed to perform the final editing stage in the compressed domain; exactly how this is to be achieved remains an open topic of research.

The editing application as described above performs the edit conforming step in real time and only allows a single user to access the—currently very expensive—switch at once. There are a number of possible extensions to the application that would greatly enhance its flexibility. The actual switching functionality of the ATLANTIC switch is normally only required during the transition between two clips; for the rest of the time the switch merely acts as a transparent re-coder. To maximise the useful throughput of the switch it could be removed from the signal path until it was actually needed, freeing it for concurrent use by other users’ editing tasks. With suitable scheduling of access to the switch, each user’s programme could still be generated in real time. Indeed, with a suitably powerful switch it might well be feasible to run edit decision lists faster than real-time. Alternatively, if real-time edit conforming was not required (as is probably the case if the programme is just going to be stored on a server), the switch could equally well be implemented in software. If the number of edits in a given programme was not excessive it should still be possible to produce the finished programme in less than its eventual play time—the material between switch points can be generated as quickly as it can be copied from the main server to the finished programme server.

In some situations—for instance, a live news or sports programme—it may not be necessary or possible to store the finished stream prior to broadcasting it, although it may well need to be archived in parallel with the broadcast. In such cases the edit decision list could simply be run at broadcast time, provided that a real-time switch is available. Scheduling access to the switch then becomes critically important, as no delay or corruption of the output stream can be tolerated. Nevertheless, it should still be possible to share the switching hardware between multiple programmes, with the broadcast stream given priority access to the switch.

The high-level requirements of the editing application are broadly similar to those of the remote surveillance system, tempered by the fact that the editing system does not have to operate across a wide area and is entirely controlled by a single organisation. On the other hand, the quality and real-time constraints of a digital television system are even stricter than in the surveillance situation—some real-world data are given in Table 3.1. Also, the resource requirements of the editing application are at least an order of magnitude beyond the surveillance system. The need for reliable and seamless reconfiguration becomes even more apparent if one considers that the editing application may be just one part of a larger “programme production” system and that its basic components are likely to be used in ways that were not necessarily anticipated by their designers. For example, the ATLANTIC switching technology is already being used for continuity/opt-out switching and dynamic real-time transcoding (recoding with a modified bit rate or other compression parameters) [TW97]; no doubt other application areas will emerge as the technology matures. The following are some of the key application-level requirements of the editing system, contrasted
with those presented earlier for the surveillance application:

- **Heterogeneity.** This is not such a significant issue for the editing system itself, which is owned and controlled by a single organisation, and operated at a single site using hardware and software installed specifically for the job. However, in the context of the larger BBC organisation and the long-term goal of using compressed video throughout the production and broadcast chain, it is important that the application remains compatible with other systems that may be deployed later. Interoperability issues will also arise when the production system needs to interact with external content providers or outside broadcast facilities.

- **Real-time.** The timing requirements of a broadcast stream are quite strict (see Table 3.1). Many in-studio applications, especially interactive scenarios such as interviews, also need tightly controlled real-time behaviour. These constraints are easily met when every video stream in the studio is carried on a dedicated cable in analogue or SMPTE Rec. 601 format. If, however, streams are transported over a shared LAN, the network must guarantee that each stream receives its required bandwidth and bounds on latency, jitter and error rate. Clearly, if none of the streams flowing into or out of the editing system are live (that is, coming from a live source or destined directly for broadcast) there is no particular need to transport the streams in real-time. Even in this case though, scheduling access to shared hardware, and the need to synchronise audio and video streams, still impose some real-time constraints.

- **Configurability.** MPEG switching is a perfect example of the need for controlled, seamless reconfiguration. The switch operation must be scheduled so that the required frames of the input stream(s) reach the switch at the appropriate time, otherwise desired material may be lost. The switching component itself must perform certain actions before and after the nominal reconfiguration time to avoid producing corrupted data. Additionally, like the surveillance system, the editing application must be able to cope with a continually changing set of users.

- **Resilience.** Although it is unlikely to put lives or property at risk, a failure of the editing system may have serious financial consequences. So, like the surveillance system, the application must be robust in the face of failures and maintain its integrity despite unsuccessful reconfiguration attempts.

- **Availability.** Again, in common with the surveillance system the editing application will need to be available at any time; indeed, the overall programme production or studio control system is likely to be running and producing broadcast output 24 hours a day. Thus, this system should also support online upgrades and reconfigurations of its structure without disruption to ongoing operations.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Production Streams</th>
<th>Broadcast Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data rate</td>
<td>270 Mbit/s raw</td>
<td>5 Mbit/s video</td>
</tr>
<tr>
<td></td>
<td>20 Mbit/s compressed</td>
<td>256 Kbit/s audio</td>
</tr>
<tr>
<td>Coding scheme</td>
<td>MPEG-2</td>
<td>MPEG-2</td>
</tr>
<tr>
<td></td>
<td>IBIBIBIBIB...</td>
<td>BBIBPPBBPPBB...</td>
</tr>
<tr>
<td>Coder latency</td>
<td>0.1–0.5 s</td>
<td>0.1–0.5 s</td>
</tr>
<tr>
<td>End-to-end latency</td>
<td>0.1–1 s</td>
<td>1 s</td>
</tr>
<tr>
<td>Switching latency</td>
<td>0.04 s (1 frame)</td>
<td>0.04 s (1 frame)</td>
</tr>
</tbody>
</table>

Table 3.1: Typical stream parameters for digital television.

- **Security.** Security is an important requirement even in a system controlled entirely by a single organisation. For example, users may not all be equally trusted, and only certain qualified users will be permitted to use particular—typically expensive or delicate—equipment.

**Detailed requirements from the BBC**

Table 3.1 shows the typical coding and QoS parameters used by the BBC for production and broadcast MPEG streams [Tud99]. Note that these are the configurations for actual programme production and transmission; the experimental ATLANTIC system may use different settings. The table reflects the quality/resource tradeoffs that can be made in this environment. For example, in-studio production work demands the best possible picture quality, so high-bandwidth streams are used, with a corresponding increase in resource costs. On the other hand, most viewers will not have studio-quality equipment in their homes and would not benefit from receiving production-quality streams, so the broadcast stream quality can be set to the lowest level that is acceptable to viewers. Reducing the broadcast stream quality also allows more programmes to be packed into a fixed amount of transmission bandwidth. Similarly, viewers are unlikely to notice the one second transmission latency of a broadcast stream (although this delay would be perceptible to viewers participating in, say, a live phone-in programme). A technician in the studio, however, will not want to wait for a full second between starting playback of a stream and seeing the output on her monitor; to meet her demands a more costly low-latency codec must be deployed.

The MPEG coding schemes used by the two stream types are also indicative of their differing latency requirements. I-frames occur only once every 12 frames in the broadcast coding scheme, that is, every 0.5s. Therefore a viewer changing channels may observe a significant pause while his decoder waits for an I-frame to synchronise with the new channel and starts decoding the stream. Such behaviour would clearly be unacceptable in the studio environment, so a less efficient coding scheme is used where I-frames make up every second frame of the stream. The final "switching latency" parameter refers to switching the input stream of an encoder—in both cases this is required to be frame-accurate, so that if a switch is initiated during frame $N$, then frame $N + 1$ will be the first
frame derived from the new source stream. There may be a relatively long delay before viewers actually see the switch occurring. Nevertheless, the temporal constraints on the switching operation are maintained as long as every downstream point observes the switch as happening between the same two frames.

3.2 Properties of distributed multimedia applications

From the application scenarios presented above and from the literature (see Chapter 2) I have formulated a set of fundamental properties that are shared, in varying degrees, by all distributed multimedia systems. Some of these properties may seem obvious or trivial; however, I feel it is important to present them all as a unit before addressing the programming, system support and configuration requirements of these systems.

Distribution

Clearly the target applications will be distributed, over a heterogeneous system controlled by multiple organisations in the general case. The scale of the distribution ranges from a high speed single-building LAN through wireless networks to the worldwide Internet.

Continuous media

The use of continuous media streams is the major factor setting multimedia applications apart from other distributed systems. The handling requirements of continuous media data are unique; for instance, it is often acceptable to lose a small proportion of the frames in a video stream, as long as the rest of the sequence is played out on time and in the correct order.

Real-time

As a corollary to the previous point, continuous media streams are real-time data, with well defined temporal relationships both within and between streams. Therefore, the applications generating, processing and consuming such streams must also operate in real time.

Resource hungry

Again as a consequence of using continuous media streams, multimedia applications make significant demands on the resources provided by their underlying operating systems and networks. This is especially true in the case of network bandwidth and host CPU cycles, both of which are likely to become scarce resources if any significant amount of media processing is taking place.
Long lived

I anticipate that many multimedia systems, in common with other mission-critical distributed systems, will be expected to remain running reliably for extended periods of time, perhaps months or even years. Users of an application are not necessarily aware of its long-lived nature; for example, a customer of a video on demand service sees only the transient connection between her television and the video server while she is viewing a movie. The server, on the other hand, remains running continuously, delivering movies to an ever-changing mix of clients. Similar statements can be made about the application scenarios presented above.

Frequent reconfiguration

It follows from the previous point that such applications are likely to undergo numerous reconfigurations during their lifetimes, which may occur in response to user requests or external events such as network congestion. Applications supporting multiple concurrent users may be required to deal with simultaneous—and possibly conflicting—reconfiguration requests.

Temporal, structural and resource constraints

Continuous media streams impose temporal constraints on an application’s processing activities. The need to support distribution and frequent reconfiguration without shutting down the system constrains the structure—and changes to the structure—of the application. Similarly, the heavy resource requirements of continuous media streams place a lower bound on its resource usage. An upper limit on resource usage (below the physical limits of the underlying hardware) may also be imposed if hosts or network links are shared between multiple applications, or between multiple dynamically instantiated streams.

3.3 Software requirements for distributed multimedia

This section presents a comprehensive set of requirements for a system supporting distributed, real-time, continuous-media applications. Such a system is essentially a middleware layer, providing a domain-specific interface between applications and low-level operating systems and network services. Thus, this section will address the design requirements for a distributed multimedia programming framework, the functional requirements of the middleware layer supporting applications at runtime, and the unique requirements imposed on operating systems and networks by this layer. The requirements were derived through analysis of existing programming frameworks and multimedia middleware systems—including the application scenarios in section 3.1—and from discussions with researchers and professionals building and using distributed multimedia systems in the "real world".
The requirements are presented at three different levels. Section 3.3.1 lists the basic high-level design requirements for a multimedia programming framework supporting highly reconfigurable applications. Section 3.3.2 describes the features that should be provided by the programming model presented to applications, while Section 3.3.3 discusses the support requirements imposed on the underlying hardware and software platform.

3.3.1 Framework design requirements

This section presents a set of general design requirements for a multimedia programming framework, largely based on those proposed by Gibbs & Tsichritzis for their framework in [GT95] (see Section 2.1). These are extended to reflect the particular direction of this research, towards highly configurable mission-critical multimedia systems.

Economy of concepts

A multimedia framework should be based on a small number of concepts, otherwise there is a danger of the framework becoming a maze of media-specific details. It is particularly important to identify any generic concepts that apply across all media types.

Open architecture

It should be possible to extend the framework to incorporate new media types, new data representations and new hardware capabilities as they become available, without breaking existing applications that use the framework.

Distribution

The framework should help partition applications in a way that facilitates distribution. In particular, the objects within the framework should correspond to easy-to-distribute units or subsystems. Most future multimedia applications are likely to be distributed, so the utility of a framework is greatly diminished if it conflicts with the need for distribution.

High-level interfaces

A framework should provide high-level interfaces for media synchronisation, media composition, device control, database integration and concurrent media processing activities. These operations are central to multimedia programming. High-level interfaces—coupled with a support layer to provide the required functionality on a given platform—liberate the programmer from the burden of managing or implementing low-level platform-specific multimedia services. The low-level services
are in any case shared by all applications, so should be implemented once and encapsulated within an abstract platform-independent interface.

Generic

The framework should not be tied to or biased towards any one application or class of applications within the target domain. A suitably generic framework will be flexible enough to support classes of applications not specifically considered by its designers.

Portable

Programmers can no longer assume that applications will always be running on particular hardware, operating systems or networks. This is especially true given the rise in popularity of languages such as Java which allow the same code to be run (almost) anywhere without recompilation. The framework and runtime middleware should be platform independent, in that applications written for the framework are able to run without modification on any platform support by the middleware. This means that the framework must provide platform independent abstractions to the programmer and avoid making assumptions about the presence or absence of any platform specific features. Portability is clearly desirable in applications such as the surveillance system with multiple administrative domains—ideally the same application components should run unchanged on both systems. Note that programmers may still build platform specific applications or components if desired, to make use of specialised hardware that only exists on a particular machine, for instance.

Reconfigurable

I am proposing to support long-lived applications that can be reconfigured and extended while they are running, without any need to shut down and restart the application. Reconfiguration should be invisible to unaffected parts of the overall distributed system: other applications, and indeed parts of the same application that are not themselves modified by the reconfiguration, should be unaware that anything has changed. Requirements for supporting reconfiguration are discussed in detail in Section 3.4.

The final three requirements are not specifically addressed by this thesis, but are included for the sake of completeness.

Queryable

A multimedia framework should specify interfaces for querying environments concerning their capabilities. Applications can then recognise missing functionality and adapt their behaviour accordingly.
Scalable

An ideal framework would be scalable over several orders of magnitude—from a single user running an application on his local network to Internet-based systems spanning the entire planet, with thousands of users. Scalability is, of course, an issue for all distributed systems and the subject of a considerable body of literature (See [CDK94b] for further references on this subject).

Scalable representations

A multimedia framework should support scalable and extensible media representations. Once media are in digital form, improvements in quality and capabilities are tied to advances in hardware and software technology. For example, the current generation of processing units in personal workstations might be able to encode MPEG streams at half-screen resolution and up to 15 frames per second, while the next generation may be capable of encoding full-screen video at 25 frames per second. If scalable representations are used, applications can increase quality as platform performance improves, or use the representation of a stream that best matches their abilities.\(^2\)

3.3.2 Programming model

The following are the most important features of the programming model provided by the multimedia framework and supported by the runtime middleware.

Separation of concerns

Application programmers should be able to build systems using the framework without necessarily needing to know all of the low-level details of how media streams are generated, transported, processed and consumed. That is, programmers should be allowed to concentrate on what their applications are doing, rather than precisely how this is achieved. This is of course the reason for providing a middleware layer rather than expecting programmers to interact directly with the operating system. Therefore, details of hardware devices, operating systems, network protocols and media formats should be hidden as much as possible from the programmer.

Application composition

A programmer designing a complex application is likely to view the system at a much more abstract level than she would with a smaller application. The framework should allow simple components

\(^2\)The MPEG-2 standard is particularly well-equipped in this respect: it allows a video stream to be encoded at multiple resolutions and frame rates, with all of the representations bundled into a single \textit{system stream}, described in Part 1 of the MPEG-2 standard [ISO95]. An application can then decode the stream at the maximum size and frame rate it can handle.
to be composed into larger, more complex components which then be used to build applications without the programmer necessarily knowing about their internal structure. This approach encourages the use of libraries of opaque, reusable components and allows components to be replaced and upgraded without disruption to the rest of the system as long as the component’s interfaces remain unchanged.

Taking the previous point a stage further, entire applications could be composed using a high-level specification language as discussed in Section 2.2. A specification language would clearly be a useful facility for building and maintaining multimedia applications; I did in fact consider following this approach in the early stages of my research. However, since the main focus of the work has turned out to be the structuring and scheduling of reconfiguration actions, the language approach—perhaps unfortunately—never progressed beyond the status of an interesting idea. Designing a language to capture and specify all of the dynamic aspects of reconfigurable multimedia systems would be a major challenge, but one well worth pursuing in the future.

Soft real-time

Continuous media streams must often be processed in real-time; that is, there are temporal constraints on the processing of each element of the stream with respect to the passage of physical “wall clock” time. Constraints will generally take the form of upper bounds on end-to-end latency (delay) and jitter (variation in latency). Multimedia applications are usually considered to be soft real-time systems [KV93, WR93], in that a certain small amount of data can be delivered late or out of order, or lost entirely, without affecting the overall integrity of the application; This tolerance to violations of real-time constraints is application specific. The framework should allow applications to specify arbitrary real-time constraints on their streams and provide a mechanism for monitoring real-time performance at runtime.

Quality of service model

Most media streams will require some minimum amount of system and network resources—such as CPU bandwidth, physical memory or exclusive access to a particular piece of hardware—in order to be processed correctly. Resource requirements are stream and application-specific; to support these differing requirements the framework should include a Quality of Service (QoS) model that describes how resource requirements can be specified in a high-level fashion; the middleware layer must then provide a mechanism to translate these specifications into quantities of physical resources at runtime.

The cornerstone of a QoS management system is an admission control mechanism which is responsible for evaluating each application’s statement of its resource requirements and accepting or rejecting the application based on resource availability. Admission control should be performed before
the application attempts any potentially expensive allocation of resources that may not ultimately succeed. Applications accepted by the admission control mechanism will receive a guaranteed allocation of resources; those rejected may continue to run, but must do so without real-time guarantees. Admission control must be supported by detailed knowledge of the resources available at any point in time and a mechanism for enforcing upper bounds on applications' resource usage.

Data and control flow model

In general, application programmers will not be concerned with the implementation details of the components they use. However, these issues are important to the programmer building new system components. The framework needs to specify a coherent set of policies for the operation of its components to ensure that all parts of the system can interoperate. In particular, policies governing the transfer of control and data between components must be well defined. The data transfer model is concerned with issues such as buffering, flow control and whether communication is initiated by the sender of receiver(s) of a continuous media stream. The control transfer model, on the other hand, addresses the questions of how many separate threads of control are required to support a given application, and how and when is control to be transferred between these threads.

Synchronisation

The processing of continuous media streams needs to be synchronised at several different levels. Intra-stream synchronisation maintains the correspondence between the presentation of stream data and the passage of real time [JSN95]. Correct internal synchronisation is automatic as long the stream is processed within its real-time constraints. Inter-stream synchronisation is concerned with maintaining temporal relationships between two or more streams, whose processing may be distributed across multiple locations. These relationships may be natural—for example, “lip-sync” of audio and video streams—or contrived, depending upon the content of the streams and their intended use. Thus requirements for inter-stream synchronisation are generally application-specific [AH91]. Event-based synchronisation [Ste92] operates at a higher level, dealing with the coordination of activities described by stream- or user-defined asynchronous events. This aspect of synchronisation is closely related to the reconfiguration mechanisms discussed in Chapters 5 and 6.

Security

Security is a concern in any distributed system, especially one that crosses organisational boundaries as many multimedia applications are likely to do. While the algorithms and techniques for providing security are outside the scope of this work it is advisable to consider how they might be integrated into the design of the framework. The main security related issues are authentication, privacy and integrity [CDK94d] all of which are relevant to multimedia applications. A complete
framework should offer at least the ability for users to authenticate themselves and the option of encrypted communications where appropriate.

3.3.3 System support requirements

The functional requirements of the middleware make significant demands on the services provided by the hardware, operating systems and networks of the underlying distributed system. Unsurprisingly, this primarily concerns the timing and resource guarantees required for correct processing of continuous media streams. In modern multi-user operating systems the emphasis is on giving programs the illusion of full access to the machine and sharing resources fairly amongst all users [BvL91]. This policy gives good performance in the average case but on a heavily loaded system response times will quickly degrade to unacceptable levels. True real-time operating systems, traditionally used in "hard" real-time application domains such as aerospace and industrial process control, do offer the necessary guarantees. However, these applications are typically more concerned with real-time response to events and do not have to deal with the large amounts of temporally-sensitive data found in multimedia applications. Traditional real-time systems are usually designed to ensure correct behaviour even in a worst-case situation, meaning that the available resources will be under-utilised most of the time. The most important system support requirements for multimedia are:

Operating system support

Clearly, if multimedia applications are to be given guarantees of access to sufficient resources to maintain a given level of service, different mechanisms will be required to manage and control access to the available computing and network resources. The aim of a resource management subsystem in this environment is first and foremost to ensure that applications receive sufficient resources to maintain their desired QoS level. Providing a "fair" allocation of resources to each application is quite explicitly not a goal of this system; the distribution of resources amongst applications will often be deliberately unfair. For instance, it would generally be appropriate to assign a greater proportion of available resources to an application playing a live video stream than to one playing video from a pre-recorded file, since any frames lost by the live player would be irrevocably lost whereas a stored frame can always be replayed.

As well as managing resource allocation, the operating system must schedule each application’s access to its reserved resources and ensure that they are available when required. This means not only scheduling CPU use, but also access to I/O devices and memory. For example, if data are allowed to be paged out in a virtual memory environment, the system must ensure that the data are brought back into primary memory before they are needed by a component. Otherwise the component is likely to miss its processing deadline while waiting for the data to be paged in.
Network support

Distributed applications will also require real-time support from the networks connecting their components. Specifically, an application will expect to be able to deliver a certain quantity of data within a bounded time interval. The communication channel must therefore provide enough bandwidth and sufficiently low end-to-end latency to achieve this goal. Some networking technologies, such as ATM are able to make the necessary guarantees, and techniques exist [AS94, YPF94, VC95, Ven97, Kos98] to bring a degree of real-time behaviour to normally non-deterministic networks such as the Ethernet. However, the TCP/IP protocol suite—and its biggest user, the global Internet—was not designed with real-time traffic in mind and provides almost no support for real-time operation, although efforts are underway to rectify this situation [ZDE+93, YB94, GIBS97]. A network may also offer other guarantees such as bounds on error rates or jitter (the variation in latency).

3.4 Requirements for reconfiguration

The ability to reconfigure a running system without shutting it down completely is an important requirement for many long-lived or highly dynamic applications. In the multimedia domain, reconfiguration may occur in response to user actions, as an adaptation mechanism responding to changes in network traffic conditions or resource availability or as a means of upgrading the system to use new hardware or software components. I have identified a number of important requirements for reconstructions of distributed continuous-media applications. As with the general multimedia requirements in the previous section, these were developed from analysis of sample application scenarios and discussions with potential users of such applications.

Timeliness

Reconfigurations must be completed quickly; within a time interval specified by the user or application. For example, for the remote surveillance application to be functioning correctly, it is essential that a video stream is displayed promptly once it has been selected by the user. Likewise, when switching between two video streams in a television studio, the switch must take place within a known, short, interval once it has been initiated by the operator.

Atomicity

Most reconfigurations will involve changes to multiple components of the application. For example, even in the simple case of switching between two streams in the studio, it may be necessary to start the new stream playing from an archive server and shut down the camera providing the old stream.
as well as rearranging the network connections into the switch. A reconfiguration can only be considered to have succeeded if all of the changes it requires were completed successfully. Conversely, if a reconfiguration should fail for some reason, the application should remain in the state it was in before the reconfiguration was attempted. Any other behaviour might leave the application in an undefined intermediate state with the reconfiguration only partly complete.

Isolation

As a corollary to the atomicity requirement, the results of a reconfiguration should not be visible to the rest of the system until the reconfiguration has completed successfully. That is, the application should appear to move instantly to its new configuration, with no intermediate states visible. It follows from this that multiple parallel reconfigurations must be prevented from interfering with each other's actions, implying that a reconfiguration may fail if it attempts to modify an application component that has already been changed by another in-progress reconfiguration. The failed reconfiguration may of course be retried once the other has completed.

Data consistency

A multimedia application must be prepared to undergo reconfiguration while still processing continuous media streams in real time. The act of performing the reconfiguration must not cause stream data to be lost, corrupted or otherwise mishandled. In practice, this means that the end result of the reconfiguration must be a consistent system, conforming to a set of rules that describe a valid application. The notion of data consistency is also concerned with the intermediate states of the reconfiguration. In general, these will not constitute valid, consistent systems; thus, stream data should not be allowed to flow through any system other than those existing at the beginning and end of the reconfiguration.

Temporal consistency

The previous point addresses the data content of continuous media streams during a reconfiguration; these streams also have temporal characteristics that must be maintained. A reconfiguration that causes violation of stream timing constraints can be considered to have failed, even if no data has actually been lost. The distinction between data and temporal consistency can be subtle: for example, a video frame that was received but discarded because it had arrived late—perhaps due to a transient resource overload during reconfiguration—is a violation of temporal consistency. Data integrity would be lost if the frame was delivered, on time, to a component that did not yet exist or was not yet connected to the application. Streams may have different temporal characteristics before and after a reconfiguration; it is important to correctly identify the point in time where the change occurs for a given stream.
3.5 Summary

This chapter has discussed the requirements imposed by dynamically reconfigurable, distributed multimedia systems at three different levels:

1. A programming framework and associated runtime middleware layer. The framework provides multimedia-specific programming interfaces to applications while the middleware supplies runtime support for the services offered by the framework. The middleware forms the "glue" between applications and the low-level facilities of operating systems and networks.

2. Low-level support demands made on the underlying hardware, operating systems and network protocols, particularly related to real-time processing of continuous media streams.

3. Specific requirements for online reconfiguration of running applications. These were derived from two sample application scenarios based on real systems.

The analysis of the sample applications led to the formulation of a set of generic properties common to these two applications and, I believe, most other distributed multimedia systems. These applications process continuous media streams subject to real-time constraints; they often require significant amounts of system resources in order to do their jobs. Furthermore, interactive multi-user applications will undergo frequent reconfiguration. Mission-critical systems must continue to meet their real-time guarantees as well as maintain consistent application structure, even in the face of simultaneous conflicting reconfiguration requests. Section 3.4 specifically addressed the requirement for reconfiguration support.

Chapter 4 presents the basic architecture of DJINN, a distributed multimedia programming framework and runtime support package designed to meet many of the requirements discussed above. DJINN is based around the idea of modelling multimedia applications, in particular their QoS and reconfiguration properties. My approach to modelling and executing time-sensitive reconfigurations is examined further in Chapters 5 and 6.
Chapter 4

The DJINN Architecture

DJINN is an object-oriented middleware framework supporting the development and execution of distributed, component-based, continuous-media applications. This chapter presents the fundamental ideas of the DJINN framework, specifically those parts of the architecture permitting distributed applications to be composed and executed. The other novel aspects of the architecture—namely support for reconfiguration and quality of service provision—are explicitly not addressed by this chapter. Reconfiguration support is the subject of Chapters 5 and 6 of this dissertation and the QoS architecture is discussed in various other DJINN project publications [MNCK98, NKMC98, MNCK99].

At the lowest level, DJINN applications are structured as networks of active media handling components interconnected through their ports. As discussed in Section 2.1, this component based paradigm is a common feature of numerous other multimedia programming frameworks; the design of DJINN has been heavily influenced by many of these. DJINN's main contribution is in its use of a layered component architecture, with a separate model of the application that is also constructed from interconnected components. The application model separates the low-level concerns of media handling from application control and management, it forms the basic platform for DJINN's reconfiguration and QoS support and provides several other advantages that will be discussed fully below.

The remainder of this chapter is structured as follows:

- Section 4.1 motivates the use of a middleware framework for supporting multimedia systems in general and presents the design goals of the DJINN framework in particular.
- Section 4.2 presents the conceptual architecture of the framework. This section discusses the benefits and drawbacks of the component-based architecture in general and DJINN's layered implementation in particular. The extensions to the architecture to support high-level structuring and application composition are also described in this section.
• Section 4.3 describes the current prototype implementation of DJINN with reference to the conceptual architecture in the previous section. My implementation to date has primarily used the Java programming language with Java RMI [Sun97c] for distribution support; this section will discuss some alternative implementation strategies that have arisen out of my experience with DJINN so far.

• Section 4.4 returns to the application scenarios first presented in Chapter 3 and illustrates how the sample applications might be constructed and executed using DJINN.

• Section 4.5 discusses some of the benefits and drawbacks of the DJINN architecture in greater detail.

4.1 Motivation

Previous chapters have motivated the concept of a multimedia programming framework, and developed a comprehensive set of requirements for such a framework. As the earlier requirements analysis shows, there exists a wide range of multimedia application classes that in spite of their difference, nevertheless all share certain basic characteristics such as their use of real-time continuous media streams. Thus, it seems reasonable to provide generic support for the shared properties of multimedia systems. Such support must address not only programming issues—provision of abstractions and interfaces for common multimedia programming tasks, for instance—but also the realisation of those abstractions at runtime.

At the outset of this research, the broad design goal of the DJINN project was to “build a framework suitable for investigating the system support requirements of distributed multimedia systems.” The early stages of the work, carried out primarily by myself and Hani Naguib, were heavily influenced by the Gibbs and Tsichritzis multimedia framework described in [GT95] which was—most encouragingly—actually implemented, with some substantial applications built and documented, and which went to considerable lengths to achieve a generic, flexible programming platform. Many of the concepts now central to DJINN derive directly from those found in the “G&T” framework.

However, while the strengths of the G&T framework lie in its powerful basic abstractions and comprehensive class hierarchies of media formats and devices, it lacks explicit support for building distributed applications and does not address any QoS issues. In addition, the authors’ description of the framework is very much oriented towards the application programmer; they do not discuss in detail the provision of operating system and network level support for applications built using their framework. Because of these missing features, the unaugmented G&T framework was not suitable as a platform for our investigations. Therefore, we elected to use the fundamental concepts realised by G&T as the foundation for our own multimedia programming framework, built from the ground up, with the additional goals of:
Support for construction and runtime modification of distributed applications.

Support for QoS management—the ideas for application modelling and later QoS-controlled reconfiguration developed from this goal.

Specifically addressing the requirements imposed by our applications on the underlying (real-time) infrastructure.

Flexibility in application construction, allowing for rapid prototyping and experimentation.

Support for "generic" applications, independent of particular hardware and system software configurations.

### 4.2 Conceptual architecture

Figure 4.1 shows the main components of the DJINN architecture. The darker shaded parts of the diagram are covered by this chapter; those with lighter shading are addressed by Chapters 5 and 6. In keeping with its G&T origins, DJINN is an "object-oriented" framework. This means that the architecture has been designed according to object-oriented principles and that most of the DJINN entities described below will be most naturally represented by objects, in the usual object-oriented sense of the term. I have assumed that the reader is familiar with basic object-oriented terminology and design principles.
4.2.1 Components and ports

Components and ports are the fundamental building blocks of DJJNN applications. My definitions of these entities are derived in part from those presented by Gibbs and Tschritzis in [GT95]:

Component. In the general case, a component is a semi-autonomous, active media-processing entity. The set of all components may be further subdivided into the three mutually disjoint and inclusive subsets of producers, consumers and transformers of media data. Components are considered to be only semi-autonomous because, while they can act on their own initiative—for instance to modify their behaviour in response to received media data—they are still ultimately under the control of an outside entity, which might be another component or the application program itself. The term active refers to the ability of a component to have its own internal thread(s) of control, such that it can process data concurrently with other components of the system. Not every component will be active in this sense, but it is necessary to have at least one true active component in an application, to drive the flow of media data.

Port. Ports provide the means for components to communicate with one another; they are the entry and exit points for media data streams flowing into and out of components. Each component has zero or more ports attached to it, with the ports said to be “owned by” the component. Components with zero ports are both legal and useful—see the discussion of composite components in Section 4.2.4. Ports are unidirectional; that is, each port is further classified as an input or output port according to the direction of data flow. Furthermore, ports are typed according to the type of media data they will carry.

Components can be classified as producers, consumers or transformers by the mixture of ports they possess:

- **Producers** have only output ports.
- **Consumers** have only input ports.
- **Transformers** have at least one input and one output port.

Components are linked together by making a connection between an output port of one component and an input port of another component. Media data can then be transferred between the two ports and their attached component(s).

- The types of the connected ports must be compatible; that is, the type of the input port must be the same as, or more general than, the type of the output port.
For example, an input port typed as being able to receive uncompressed video could be connected to an output port producing raw RGB video but not to a port that produced MPEG compressed video. There are no further restrictions on the the ports being connected, so that connections may be made between ports on the same component or otherwise arranged so as to form a cycle in the media data flow.

- Data transfer is *source-driven*: "upstream" components deliver data to their "downstream" neighbours as they choose; there is no need for the downstream component to request new data.

It would of course be possible to have a consuming component signal a producing component—using a separate "control" channel—to trigger the delivery of data in a demand-driven fashion, although DJINN has no standard mechanism to achieve this.

In keeping with my stated goal of flexibility I have made the assumption that, in general:

- A component will *not* know the identity of the other components it is directly or indirectly connected to, nor will it necessarily be able to discover this information.

The component knows that some entity is supplying data to its input port(s), and that some other entity is presumably receiving the data it sends to its output port(s); the identity of these other entities is irrelevant. Thus, in the general case a consuming component will have no idea where to send a request for demand-driven data, so that such schemes cannot easily be implemented without specific application knowledge. Clearly, there are other situations where knowledge of component connectivity is useful; DJINN's "path" objects (Section 4.2.5) encapsulate end-to-end connectivity information as required by each application.

Figure 4.2 shows a simple two-component application with a variety of inter-component connections.

### 4.2.2 Connectors

Again following the Gibbs and Tschritzis model, direct inter-component connections can only be made between ports residing on the same physical host or, more precisely, within the same address space.

- Distributed applications are constructed using a specialised component subclass, the *connector*.

A connector is defined as follows:
Connector. A connector is a component that provides media data transport across physical or virtual machine or address space boundaries. Connectors will generally be implemented as fragmented objects, with a separate physical object instantiated on either side of the boundary and a private communication channel between the two, although the user of the connector need not necessarily be aware of this. From the point of view of other components in the same address space as a connector endpoint, the source end of the connector appears to be a consumer component while the sink end behaves as a producer component. This is illustrated in Figure 4.3. The nature of the communication channel between the two endpoints is specific to the particular type of connector. A multicast connector is created by instantiating multiple sink endpoints.

DJINN's use of connectors differs somewhat from that described by Gibbs and Tischritzis. In their original work, connectors are separate class of objects distinct from components, and a connector is required between every pair of connected ports, even within the same address space. DJINN has optimised this model by observing that in most cases the connector between components in the same address space was unnecessary and inefficient, simply adding a redundant object and
the overhead of additional method invocations to the flow of media data between the components. In situations where a "local" connector is required—such as to provide buffering or rate control between a camera and an MPEG compressor—a suitable connector can be explicitly instantiated by the application. Without compulsory connectors between components, it becomes necessary to provide direct input port to output port connections, as in DJINN. Once direct connections are available, it is a trivial step to make all connections this way and to transform connectors into components with their own input and output ports. This is similar to the approach taken by the Medusa framework (Section 2.1.4) and offers, I believe, several advantages:

- **Efficiency.** There is no need for media data to pass through an arbitrary number of "do nothing" connectors on its journey from producer to consumer.

- **Clearer application structure.** Connectors are only present where they are actually needed to achieve the processing requirements of the application.

- **More logical design.** Connectors share many of the properties of transformer components—they accept a stream of media data and produce another, possibly different stream—so it makes sense to group them together with other transformers.

One might think that similar arguments could be made for the elimination of ports—after all, ports do not perform any processing themselves and simply add more method-invocation overhead to the data flow between producer and consumer. However, there are sound reasons for retaining ports as separate entities: ports provide a common, externally visible interface to the possible connections to a given component, as well as hiding the details of inter-component connectivity from the components themselves. If ports (or an equivalent mechanism) were not present, component-specific connectivity interfaces would no doubt proliferate, leading to incompatibility between components and obfuscating application structure. There is perhaps a case to be made for folding the functionality of ports directly into the component object, to reduce the method-invocation overhead of the two port objects involved in every connection, while still retaining the logical notion of ports as distinct from components. This is an optimisation that might be considered during a future re-design of the DJINN architecture.

### 4.2.3 Application modelling

The simple architecture of components, connectors and ports described above is adequate for constructing simple local and distributed applications. However, this system is likely to quickly become error-prone and unmanageable in the face of a large, complex application such as the digital television studio, with thousands of components spread across multiple administrative domains and controlled by numerous users. The major part of the problem is the very low-level view of the system offered by the basic port-and-component architecture: every entity in an application is visible
simultaneously, with no attempt to impose a higher-level structure. Additionally, the basic architecture has no notion of end-to-end streams, or indeed any other relationship between components; it also offers no support for QoS management or controlled reconfiguration.

My approach to meeting the requirements presented in Chapter 3 is based around the use of a dynamic runtime model of component-based multimedia applications. The ultimate intention of the application model is to encapsulate all relevant aspects of the structural configuration, reconfiguration and QoS properties of the application. The model is itself built from interconnected components, so that DJINN applications have a two-layer structure as shown in Figure 4.4. An application is thus composed from both model and active components and ports:

**Active component.** The active components of an application are semi-autonomous active objects that produce, consume and transform multimedia data streams; that is, they are just the basic components described above. Active components are distributed so as to meet the processing requirements of the application. In general, this means that they must be co-located with the multimedia hardware they control. The set of active components forms an entirely flat structure; that is, there is no hierarchy or composition of active components.

**Active port.** Likewise, the active ports of an application are the basic media-handling ports described above. Active ports may only be attached to active components; similarly, active components may only own active ports.

**Model component.** Unlike active components, model components do not directly process media
data and can be located wherever is convenient for the application user or programmer. Indeed, the model components of an application may be distributed entirely differently to its active components, for example in a video server system where the server and clients are controlled by different people and organisations. In addition, model components may be hierarchically composed as described in the next section.

Model port. Model ports are related to model components in the same way as active ports to active components.

The most crucial distinction to be made between active and model components is that the operations of active components are time-critical—these components are processing streams of media data in real time and must be able to respond quickly enough to meet the application's quality of service requirements. Model components never actually come into contact with any media data and thus do not have to meet the same real-time constraints as their active counterparts. This distinction has important implications for the implementation of the framework. Note that a model component may in fact be "active" in the sense of having its own internal thread(s) of control. This does not, however, transform a model component into an active component—the distinguishing feature remains that an active component directly handles media data while a model component does not.

4.2.4 Application composition

There is a well defined relationship between the model and active layers of an application. The model components of an application are arranged in a tree-structured hierarchy, where the leaves of the tree are atomic model components, each corresponding to a single active component (for example, the Video Source and Display components in Figure 4.4). Atomic model components effectively act as smart proxies or agents for their underlying active components, exporting a common interface such that all "Camera" components will offer a common set of operations irrespective of the physical type of camera controlled by the active component. Additionally, atomic model components store QoS characteristics of their underlying active components, such as the frame rate and size of the media streams being processed and their relationship to the resources consumed by the component [NKMC98]. The connectivity of the active layer is mirrored by the atomic model components: each has the same set of ports and inter-component connections as its active counterpart.

The interior nodes of the model component tree are the composite components. These components do not correspond to any one active component; rather, they encapsulate a sub-tree of the application model, with the composite component at the root of the tree. Composite components are able to make arbitrary changes to their internal structures, and are expected to export type-specific interfaces giving component users the ability to perform—in a controlled manner—some subset of these changes. For instance, a video-conferencing component would provide operations to add and remove conference participants. A composite component models the connectivity of its encapsulated
sub-tree as a directed graph that can be expanded down to the atomic model component level. Note that active layer objects cannot be composed; all composition is done in the model layer. The use of composite components leads to a number of advantages related to the structuring of applications:

- **Hiding of complexity.** Composite components hide the low-level active component structure of the application from human or program users who do not wish to see it. For example, a video-conferencing system can be treated as a set of “participant” components linked through a central multicast connector rather than as a large collection of cameras, displays, microphones, speakers and network links.

- **Increased abstraction.** In a similar way, composite components allow the process of application construction to be performed at a higher level of abstraction. Instead of building a complete application using atomic components, a designer can simply select composite components with the appropriate ports, control interfaces and behaviour, and compose the application from these.

- **High level behaviours.** In contrast to many other frameworks that provide similar abstractions, DJINN’s composite components are more than mere shells or containers around a collection of subcomponents. Composite components are first-class objects that can implement their own type-specific high-level behaviours. Most often, these will be structured manipulations of the object’s subcomponent graph, but a composite component might also, for example, offer a graphical user interface or a control interface to other non-DJINN systems.

The third point above is particularly important, as it opens the way for the provision of reconfiguration and QoS support within the component framework itself. It is also worth noting that, in common with atomic components, composite components are intended to be independent of the particular underlying media hardware in the active object layer. Thus, a video-conferencing composite component can be successfully operated wherever a suitable collection of cameras, microphones, etc. is available. A particularly intelligent component would even be able to integrate mutually incompatible components into the same application by connecting transformer components to convert the media streams to and from a common format.

**Federated models**

As I have mentioned already, the model components of a distributed DJINN application may themselves be distributed; however, this distribution is usually carried out on a very different basis to the active components in the same application. While active components are distributed so as to meet the media processing needs of the application, the distribution of the model should reflect its control and management structure. For instance, each user of a multiuser applications is likely to
want the part(s) of the application she directly controls to be located "close" to her—probably on her personal workstation or mobile terminal—to ensure the most responsive access when querying or reconfiguring the application. Conversely, she probably does not want other users to have the same level of control over "her" components.

One can then imagine the application becoming federated into a set of cooperating sub-models owned by the individual users and perhaps other more abstract entities such as a "conference manager". The members of the federation are interconnected in the usual way to compose the complete application. This is most effective when the components making up each member of the federation can be encapsulated within a single composite component that denies other members access to its internal structure (Figure 4.5).

This gives each member complete freedom in how to realise its part of the application, as long as it conforms to the interface of the enclosing composite component. Note however that a distributed application model is not necessarily a federated model: the key feature of a federation is that each member is effectively a "black box" when viewed from outside, appearing to be a single atomic component. Every member of the federation will have a different, although equally valid, view of the overall structure of the application. Likewise, federation members are more than just opaque composite components: each is an independent entity that has its own policies with regard to reconfiguration, QoS and resource management and can act on these without reference to the rest of the application. Contrast this with a non-federated application, where these functions are performed globally across the whole component tree.
It should be pointed out that DJJNN does not provide any form of access control at present, so it is not in fact possible to implement composite components that conceal their internal structure in this way. However, substantially the same effect can be achieved by generating a proxy component corresponding to each federation member's top-level composite component. The proxy exports a set of ports and methods—not necessarily the same as those offered by the "real" component—so can be treated by other federation members as if it were the original component, except that they cannot gain access to its internal structure. The proxy uses a private protocol to communicate requests and replies with the real component.

At this point it is useful to ask the question "exactly what is an application?". The ability to arbitrarily compose and federate components means that it is often difficult to absolutely define the boundaries of an "application". However, in the interest of clarity I will define an application, from the DJJNN point of view, as follows:

**Application.** That set of DJJNN model objects that is directly accessible to and under the control of a single human or machine user. Typically, but not necessarily, the set of model objects managed by a single user will be a small number—often only one—of task-specific composite components.

**Federated application.** A set of cooperating applications, as defined above. Individual members of the federation may be controlled by different users and will supply services to and/or receive services from other federation members.

Thus, in a video-conferencing system, each participant's application would consist of all the components necessary to run her end of the conference. These would assume the presence of an external conference manager entity providing access to the other participants in the system. Likewise, a video-server client would see the video server as a "black box" feeding the requested video into his own application; the server itself knows that it is serving video to many clients but does not need to know how each operates internally. Each such DJJNN application then forms part of a larger user-level application, which will usually include non-DJJNN code and data, in particular the user interface.

**Control relationship between the active and model layers**

Application programmers are in general unaware of the distinction between model and active components and in fact should never interact directly with active components. All application-level programming in DJJNN takes place at the model layer, with active components created, configured and destroyed as required under the control of the application model. As shown in Figure 4.4, the interaction between the layers uses both remote invocations and events. Remote invocations are used by the model layer to invoke operations on active layer objects. Events are used by the active
layer to provide component-specific notifications to the model layer. The notification mechanism is intended to be used to inform the application model of exceptional conditions—such as host failures or resource shortages—that the model may wish to act upon, for instance by initiating a reconfiguration.

As well as asynchronous transfer between components in different layers, events may also flow along the same paths as media streams in the active layer, interleaved with media data elements. The initial ordering of media and events within a stream is maintained. Events arriving at a component in this way, through an input port rather than from the model layer, will be handled by the component in exactly the same way. In addition, a component may forward any delivered event downstream to other connected components. This "events-within-streams" mechanism allows applications to synchronise reconfigurations with media data flow, a feature that will be explored further in Chapter 6.

### 4.2.5 Modelling end-to-end properties

The application model as described so far encapsulates the structural composition of an application, as well as its QoS and reconfiguration properties on a component-by-component basis. It is also useful to model applications in terms of end-to-end properties that must hold between media data production and consumption points, across multiple components. Most commonly these will be properties of the underlying transport layer, such as the end-to-end latency, jitter and error rate, or QoS characteristics of the media flow itself, for instance video frame rate, audio bandwidth or even just the knowledge that the media stream consumed by a particular sink is in fact the same as—or derived from—that produced by a given source (such relationships are not always obvious in complex applications with multiple interconnected producers and consumers). The common denominator amongst all of these properties is that each describes some (quantitative or qualitative) characteristic of the media flow at a particular point relative to the corresponding media flow at another point further upstream.

DJINN integrates these properties into an application model as path objects:

**Path.** A path is an end-to-end management construct describing the media data flow between a pair of endpoints chosen by the application. A path encapsulates an arbitrary sequence of ports and components that carry its data and declares the end-to-end properties of that sequence. These properties are application specific, but will generally include the overall latency, jitter and error rate of the path as well as some media-specific properties, dependent on the media type carried by the path.

It is an application’s responsibility to identify the end-to-end flows that are interesting or important to it and specify paths accordingly. The only restriction imposed by DJINN is that there must be a media-flow relationship between the specified endpoints; that is, the stream data arriving at the
Figure 4.6: Paths encapsulate end-to-end properties of applications.

downstream endpoint must either be the same as, or derived from, that departing from the upstream endpoint. Figure 4.6 shows a pair of example paths in a simple application.

The determination of whether or not any given media stream is derived from another relies on components to correctly identify the dependencies between their input and output ports: if a component indicates that its output port \( M \) "depends on" its input port \( N \), DJINN will assume that the stream produced by \( M \) is derived in some way from that consumed by \( N \). This is illustrated in Figure 4.7, where the only possible path between components \( A \) and \( B \) must pass through ports \( M \) and \( N \).

There is no valid path through \( P \) in this scenario, because the output at \( M \) does not depend on the input at \( P \). In constructing a path between two endpoints it is both necessary and sufficient that the stream at every point along the path is directly or indirectly derived from the stream at the upstream endpoint of the path. Note also that there may be more than one valid path between the specified pair of endpoints, potentially with quite different characteristics. The current implementation will simply choose the first path it finds, which may not be the most suitable, given the coarse-grained nature of the depends-on relation maintained by components. This is a recognised limitation of DJINN that I hope to remove in a future revision by either allowing the application to specify certain intermediate points that must lie on the path, or by somehow "bundling" the different routes taken by data between the two endpoints into a single path. The latter approach may require a more elaborate model of end-to-end properties since the path may have different characteristics depending upon which route the stream follows, noting that this too may change over time.

4.3 Framework implementation

The current prototype DJINN implementation consists of a library of classes from which applications can be constructed and new component classes derived, and simple distributed runtime envi-
The majority of the system is implemented using the Java [AGH00, JSG00] programming language, with the remainder—mostly time-critical media handling code in active components—using C [KR88] or C++ [Str90].

### 4.3.1 Architectural overview

Every host that wishes to run or participate in DJINN applications must be running an instance of the "DJINN runtime." This is a Java object that must be unique within a given Java Virtual Machine (JVM); that is, only one instance of the runtime class may exist within a given JVM. The runtime object, along with various helper objects that it creates, provides a number of basic services to DJINN applications:

- DJINN object creation on behalf of other runtime objects.
- Invocation of methods on DJINN objects in other address spaces.
- Object location, for both local and remote objects.
- Event delivery to remote runtime environments.
- Resource management and other QoS-related services.

DJINN objects—including components, ports, events and event dispatchers—are created in the context of a particular runtime; i.e. within a single JVM. An object may be named in several ways:
A unique identifier (UID) assigned to the object when it is created. UIDs are only unique within the assigning JVM, but can easily be transformed into a globally unique identifier by composition with the host name and process identifier of the JVM.

A non-unique textual name, initially assigned when the object is created but able to be changed at any time. This name may also be exported via a per-JVM registry, allowing it to be looked up by remote runtime objects. Names exported in this way must be unique within the JVM and cannot be changed once they have been exported.

An opaque, globally unique object reference generated by the distributed object system.

Objects can be looked up locally via their UIDs or remotely through their exported names—non-exported objects cannot be looked up in this way. Either type of lookup resolves to an object reference. Additionally, method invocations may return object references, not necessarily to exported objects.

The model layer is location-transparent, so model objects may be instantiated on any host where a DHNN runtime exists. Specifically, the model objects for a given application neither need to be co-located nor instantiated anywhere particular with respect to the active components of the same application. Nevertheless, in the interests of efficiency it may be appropriate to co-locate the entire model or place certain model objects close to their active counterparts.

No such location transparency exists for active objects; rather, these are created within an explicitly specified JVM. This is entirely legitimate, however, since active objects must generally be co-located with particular items of hardware or at strategic locations within a network. It is the responsibility of each active object's model to determine where it should be instantiated; in most cases atomic components will delegate this decision to a higher-level composite component that is able to take a global view of the application.

Communications amongst model objects, and between model objects and their active counterparts, consists of method invocations over some distributed object framework. This is currently Java RMI [Sun97c], but could equally well be CORBA, Microsoft DCOM or any other similar mechanism. Active objects, on the other hand, cannot make direct invocations on their models. Their only avenue of communication with the model is through asynchronous event notifications. These use a publish-subscribe model similar to that of the Java Abstract Window Toolkit (AWT), although the event recipients may of course be distributed. The event transport currently uses RMI, purely for convenience of implementation. Events are delivered by separate, dedicated threads such that they appear asynchronous from the point of view of both the sender and receiver.

Events as well as media data flow between active objects. Within a JVM, these are delivered by direct method invocations on the downstream object. Data flows between JVMs are managed by federated connector components, implementing some private protocol between the members of the
federation. The DJINN runtime provides support for serialisation and fragmentation/reassembly of objects passed through connectors.

The control flow model for active objects has not been formalised, but generally takes the form of a single thread per producer object; this thread transfers control to each downstream object in turn until it reaches a consumer object or JVM boundary. Exceptions to this rule occur in the case of transformer components that, for example, buffer data for a significant period, those with different input and output data rates, or that have different number of input and output ports (such as a mixer). These components will typically create their own threads to handle outgoing data, decoupled from the incoming upstream thread. Likewise, synchronisation mechanisms have not been directly addressed in the current prototype. Intra-stream and—to a lesser degree—inter-stream synchronisation are maintained automatically within the tolerance of the real-time infrastructure. It would be relatively trivial to add a rate-control feedback mechanism to path objects in the model, using statistics gather from the active objects.

Component and port objects in a DJINN application pass through a number of different states during their lifetimes, derived from those used in the G&T framework. The different states are best explained by way of the methods used to effect transitions between them:

**powerOn.** This method moves the object into a state where it is ready to immediately begin processing media data, but is not actually doing so. For a component, this will involve allocating any necessary resources, creating threads, making network connections and perhaps pre-fetching some data if this is possible. For ports, the powerOn transition is a no-op. A powered-on component may be holding exclusive access to resources, such as a camera or other media device.

**powerOff.** The inverse operation to powerOn. This moves the object back to a quiescent state where it is not holding any resources and is not able to immediately begin processing data. As with powerOn, this method is a no-op for ports.

**start.** Starts the actual processing of data through the component or port. PowerOn must have been invoked on the object before this method is called. A port will not pass any data in or out unless it has been started.

**stop.** Stops the movement of data through the component or port. This method must be called before powerOff can be invoked.

The above descriptions are obviously more directly applicable to active objects, but from the point of view of an application programmer the model objects do exhibit this behaviour when they undergo the same transitions. Because the active peer is not necessarily instantiated at the same time as its model the ModelObject class provides an additional pair of methods,
activate and deactivate, to explicitly create and destroy the object's active peer. The activate method must be invoked before powerOn and deactivate cannot be called until the powerOff method has been invoked. Figures 4.8 and 4.9 show the possible state transitions of a ModelObject and its active peer, from the point of view of the model. Additional states and transitions, such as a "paused" state, may be implemented by subclasses if necessary.

4.3.2 Implementation language

I chose to use Java as the primary implementation language for several reasons:

- DJINN was designed to be explicitly object-oriented in its approach to application construction, so it seemed appropriate to implement it using a true object-oriented language.

- Java has a certain amount of built-in support for building concurrent and distributed applications: TCP/IP networking classes, remote method invocation (RMI), dynamic class loading, threads and basic synchronisation primitives. Additionally, the Abstract Window Toolkit (AWT), a standard Java class library, allows a Java application to interact with the windowing environments on a variety of platforms without any changes to the code; potentially a useful feature when building a multimedia framework.

- Java seemed more suitable for the rapidly evolving prototype style of development that has characterised the DJINN project. The use of garbage collection rather than relying on the programmer to track memory allocations, and strong compile- and run-time type checking are useful features in this regard.

- A stable version of the Java development system had been publicly available for less than a year when work started on DJINN. At the time there was—dare I say it—a large amount of
hype and very little substance surrounding the new language (many would argue that little has changed since then). In any case, given that Java appeared to be suitable from a technical standpoint, the DJINN prototype seemed to be an ideal opportunity to evaluate the language and platform in the context of a relatively large and demanding project.

Java may seem to be an ill-advised choice for several reasons: it is an interpreted language, with all of the poor performance connotations that statement carries; it does not adequately support real-time operation, at least not on multimedia-capable hardware; and it deliberately denies programs access to the underlying hardware of the machine on which they are executing. These are valid concerns, which have been addressed by DJINN:

- Interpreted Java code is indeed known to run considerably slower than the equivalent C++ code on a given target machine, at least an order of magnitude slower in some cases. However, this gap has narrowed somewhat recently with the widespread deployment of "just in time" (JIT) compilers, which compile Java bytecode to native machine code at runtime and, more rarely, full Java to native code compilers which compile Java source code to a platform-dependent executable. In addition, performance has never been a primary goal of DJINN, at least not in its current prototype state, so I have not been unduly affected by Java's lack of speed.

- Real-time scheduling and resource guarantees are clearly key requirements for any effective continuous-media system. Despite the availability of products such as Real-Time Java
Java still cannot claim to be a convincing player in the real-time field. However, one advantage of DJINN’s two-level component architecture is that the requirement for real-time performance is almost entirely restricted to the much simpler active layer, where the actual continuous-media processing takes place. If true real-time behaviour is required, the active layer can be re-implemented, with a reasonable amount of effort, in C++ under a real-time operating system. This has in fact been done with DJINN: Hani Naguib has re-implemented the active layer as a C++ class library on the Chorus [BGG+91] real-time operating system, as part of his investigation into QoS modelling. Alternatively, if real-time behaviour is not a critical requirement, DJINN can be operated in a best-effort mode on conventional systems; for the case of lightly-loaded hosts and networks I would expect performance to be approximately the same as that obtained on a real-time system.

- Java programs run in a virtual machine environment that restricts access to the resources of the underlying hardware and operating system. However, direct access to the hardware or specialised code libraries is necessary to drive most multimedia devices. Fortunately, Java itself provides a solution to this problem in the form of “native code” libraries. These allow all or part of the code for a Java class or classes to be implemented in C or C++, compiled into a platform-dependent shared library and dynamically linked with the Java program at runtime. I have used this technique to add support for various real multimedia devices to DJINN, as well as to implement some parts of the DJINN runtime that were too slow or simply impossible in Java—these will be discussed further below.

Largely as a consequence of the decision to use Java as the implementation language, the current DJINN prototype uses Java RMI for the majority of its distribution needs. Specifically, RMI is the mechanism by which: distributed and federated parts of an application model communicate with one another; model component invoke methods of their corresponding active component; and events are delivered between the model and active layers. RMI uses a TCP-based protocol, while the transfer of continuous media data and events between active components are generally handled by more lightweight, connectionless protocols (raw UDP, IP multicast, RTP, etc.)

The most unfortunate side effect of RMI is that it is only supported by Java; meaning that the option of re-implementing the active layer in C++ is not available if RMI is used as the remote invocation mechanism. The most recent release of the Java system has added support for CORBA, allowing Java applications to interact with CORBA-compliant clients, servers and ORBs implemented in any language with CORBA bindings. Thus, a version of DJINN using CORBA as its remote invocation mechanism can operate with a C++ active layer; this is the approach used by Naguib in [NKMC98] and one which I plan to adopt in any future re-implementation of DJINN.1

1This task has been made easier by the latest release of the Java development system (JDK 1.3) which offers the ability to make RMI invocations over IIOP (the OMG’s Internet Inter-ORB Protocol). Thus, RMI clients can now access CORBA servers with minimal changes to the RMI code. However, this mechanism requires the CORBA Object-By-Value extension [OMG98] which, at the time of writing, was supported by very few non-research ORBs.
4.3.3 Real-time support assumptions

DJINN makes a number of assumptions about the capabilities of operating systems and networks it run on which, although not necessary for the system to operate and successfully run distributed applications, are important if DJINN is to achieve its goals of real-time performance and QoS support. Specifically, DJINN assumes that the host(s) where active components are instantiated are running a real-time operating system, and that the networks used to transfer continuous media data provide upper bounds on end-to-end latency, jitter and error rate. The framework also assumes that these hosts and networks allow applications to reserve CPU cycles, memory and network bandwidth, and the these reservation are guaranteed to be honoured.

Obviously, such a situation is unlikely to occur in the real world of non-deterministic operating systems and a best-effort Internet. As stated above, DJINN can operate in a best-effort mode in this situation, although it will not be able to guarantee real-time delivery of continuous media data. As most of the work presented in this dissertation is concerned with the model layer, I have not been overly concerned about the lack of true real-time support for the active layer. DJINN has, however, been demonstrated on a small-scale real-time system running the Chorus operating system.

4.3.4 Class library overview

The Java classes making up the DJINN class library and runtime are arranged into a hierarchy of six top-level packages\(^2\) as shown in Figure 4.10. The following sections outline the contents of each package, with code samples where appropriate. Note that the code snippets in these sections do not show the complete interface to each class and in most cases are not legal Java class definitions. In particular I have omitted both the code necessary to support RMI or CORBA and code to throw

\(^2\)A package is a Java construct that groups together a set of closely related classes. Members of the same package have access to each other's protected fields and methods, which is denied to members of other packages. A similar effect can be achieved through the use of the `protected` modifier in C++ classes.
and handle exceptions, in the interest of presenting the main points of the architecture as clearly as possible. The classes described here are insufficient to build a useful application: there are no components in the basic DJINN library that actually do anything interesting from a multimedia-processing point of view. Thus, these classes will primarily be used by programmers creating new classes of component; such components can then be used to build real applications.

Package djinn

This package provides the basic runtime support for DJINN applications described in Section 4.3.1. The package consists of the three classes shown in Figure 4.11; every DJINN application will inherit from or invoke methods in these classes.

DjinnObject (Figure 4.12) is the base class for almost all of the objects making up a typical DJINN application—components, ports and paths are all subclasses of DjinnObject. A DjinnObject is a remotely-invocable object with a number of DJINN-specific attributes including the object’s name and UID. The runtime method gives access to the DjinnRuntime object on the same host as the target DjinnObject. The putAttribute and getAttribute methods allow the storage and retrieval of class- or application-specific values indexed by character strings. The current RMI-based implementation of this class also exports methods (not shown here) to determine the actual implementation class of a remote DjinnObject reference and whether a remote reference in fact refers to an object located in the caller’s local JVM.

The DjinnRuntime class (Figure 4.13) is the heart of the runtime support system. A single instance of DjinnRuntime must exist in every JVM that is going to run DJINN applications. The class has no public constructors; rather, the single instance is created automatically when the class is loaded. One important function of the runtime is to maintain certain state shared between all of the DjinnObjects in the local JVM, in particular the host address and JVM identifier that are needed, along with the object’s UID, to globally identify a DjinnObject. The host and addr methods return the host address as a DNS name or raw IP address respectively, while the port method returns the IP port number used by the RMI system in the local JVM, which serves to uniquely identify the JVM relative to the host. If CORBA was to be used instead of RMI for distribution support, a different technique would be required to identify the virtual machine.
public abstract class DjinnObject {
    // Constructors
    protected DjinnObject();
    protected DjinnObject(String name);

    // Methods
    public int uid();
    public String name();
    public DjinnRuntime runtime();
    public Object getAttribute(String key);
    public Object putAttribute(String key, Object value);
}

Figure 4.12: Class djinn.DjinnObject.

public class DjinnRuntime {
    public String host();
    public InetAddress addr();
    public int port();
    public Registry registry();
    public static DjinnRuntime runtime();
    public static Object findServer(String name);
}

Figure 4.13: Class djinn.DjinnRuntime.
```java
public class DjinnFactory {
    // Constructors
    public DjinnFactory();

    // Methods
    public DjinnObject create(String type, Object[] args);
}
```

Figure 4.14: Class `Djinn.DjinnFactory`.

The remaining methods in the `DjinnRuntime` class are used to access the various runtime support objects present in the local JVM. Two particular objects with specific access methods are the runtime object itself, obtained through the static (class) method `runtime`; and the local RMI registry, which publishes the names of objects that have chosen to register with it, accessible through the `registry` method. The `findServer` method merely provides a convenient way to look up objects in the local registry—it returns any object it finds registered under the given name. Objects are not required to place themselves in the registry, but should do so if they need to be readily accessible to objects in other JVMs; such objects can then simply look up the target object using its well-known name. `DjinnRuntime` also contains methods for logging status and error messages to a terminal window, and methods that use the Java Reflection [Sun97a] package to look up methods and constructors of objects or classes.

Note that all instances of `DjinnObject` will return a reference to their local runtime through their `runtime` method; thus, once an object in another JVM has a reference to some `DjinnObject`, it can obtain a reference to that object's runtime, discover where the object is physically located and access other objects registered in the remote JVM.

A `DjinnFactory` (Figure 4.14) is an object that can be used to create new objects. An instance of `DjinnFactory` must exist in a JVM before a remote caller can create new objects there; constructing distributed `DJINN` applications would be an extremely inconvenient process without factories. By default, a single `DjinnFactory` object is created by the `DJINN` runtime and registered under a well-known name. Any remote object can then obtain a reference to the factory and use it to create objects in the local virtual machine.

The `DjinnFactory` class exports a single method: `create`. This takes the name of a class and an array of objects as arguments. If the name corresponds to an actual class that is accessible to the factory, the reflection methods of `DjinnRuntime` will be used to search for a constructor in that class that is compatible with the given `args`. If a suitable constructor is found, it will be invoked with the members of `args` as its arguments. A reference to the resulting object is passed back to the original caller.
Package \texttt{djinn.model}

Classes in this package provide support for building and configuring application models. Component programmers will use these classes—generally by creating new subclasses to extend them—when creating new composite or atomic model components. Figure 4.15 shows the class hierarchy of this package.

The \texttt{ModelObject} (Figure 4.16) class directly extends (inherits from) \texttt{DjinnObject} and plays an analogous role amongst the model classes; that is, it is the base class for all of the objects making up an application's model. The methods in this class manage all of the non-class-specific interactions between model objects and their active counterparts. \texttt{ModelObject} has a single constructor that takes as arguments the name of the new object and the name of the host where its active counterpart will (eventually) be located. Although the framework supports multiple DJINN JVMs per host, the model layer does not currently offer any way of indicating which one should be used. The
public abstract class ModelObject extends DjinnObject {
    // Constructors
    public ModelObject(String name, String host);

    // Methods
    public String activeHost();
    public InetAddress activeAddr();
    public int activeState();
    public int modelState();
    public void activate();
    public void deactivate();
    public void powerOn();
    public void powerOff();
    public void start();
    public void stop();
}

Figure 4.16: Class djinn.model.ModelObject.

active peer is not instantiated immediately because it can potentially use resources on the target host that are not yet available, or require the presence of other objects that have yet to be created. The activeHost and activeAddr methods return the location of the active object analogously to the host and addr methods of the DjinnRuntime class.

The modelState and activeState methods return an indication of the current state of the model object and its active counterpart respectively, where the states are those shown in Figures 4.8 and 4.9. The active object’s state will in general track that of the model, except that the active object may also be reported as “not created yet” (before activate has been called) or “deleted” (after deactivate has been invoked). ModelObject also implements the basic state changing methods, although these will almost certainly be overridden by subclasses.

By itself the ModelObject class is not terribly useful; it must be extended and some more specialised functionality added before it can be used to build DJINN applications. The two most commonly encountered specialisations of ModelObject are instances of the Component (Figure 4.17) and Port (Figure 4.18) classes—as the names suggest, these are the realisation of the model components and ports discussed above.

A generic Component object is little more than a holder for a set of ports, providing methods to add and remove ports, and the query the ports already attached to the component. The ports method returns an array whose elements are references to all of the ports owned by the component, while findPort looks up a port by name. AddPort checks for and rejects duplicate names, so that the ports attached to a given component are guaranteed to be uniquely named.

Port objects are always associated with a particular Component, which is specified in the constructor invocation. It is not necessary to specify a location when creating a Port, since a port
public abstract class Component extends ModelObject {
    // Constructor
    public Component(String name, String host);

    // Methods
    protected void addPort(Port port);
    protected void deletePort(Port port);
    public Port[] ports();
    public Port findPort(String name);
}

Figure 4.17: Class djinn.model.Component.

public abstract class Port extends ModelObject {
    // Constructor
    public Port(String name, Component owner);

    // Methods
    public void connectTo(Port port);
    public Port connectedTo();
    public Component locatedOn();
    public void disconnect();
    public boolean isConnected();
    public Class getMediaClass();
    public void setMediaClass(Class media);
}

Figure 4.18: Class djinn.model.Port.
public abstract class CompositeComponent extends Component {
    // Constructor
    public CompositeComponent(String name);

    // Methods
    public void add(Component comp);
    public void add(Component[] comps);
    public void remove(Component comp);
    public void remove(Component[] comps);
    public Enumeration components();
}

Figure 4.19: Class djinn.model.CompositeComponent.

is always co-located with its owning component. A newly created Port will invoke its owner’s
addPort method to indicate that it has been attached. In addition, a Port offers methods to
connect and disconnect it from other ports, set the media class it will allow to flow through it (the
port “type” discussed above) and query its location, connectivity and type.

The final class in the djinn.model package that should be mentioned is
CompositeComponent (Figure 4.19). This is a subclass of Component from which
all composite component classes will directly or indirectly inherit. Because a composite component
has no single active counterpart, it is not necessary to specify a location when invoking its construct-
or; any encapsulated atomic components will of course need to indicate their locations in the usual
way. The class provides methods for managing the set of sub-components, which may be added or
removed individually or in groups. Furthermore, CompositeComponent overrides the state
changing methods inherited from ModelObject in order to keep the state of its sub-components
synchronised. Subclasses are likely to override some, if not all, of these methods to reflect their
particular requirements.

Package djinn.event

DJINN's event subsystem is used for a variety of purposes: dissemination of notification from the
active layer to the application model; transport of control messages within the active layer; and, in
a modified form, delivery of invocations from model component to their active counterparts (see
the following section for details). The classes in package djinn.event, shown in Figure 4.20,
define the event objects themselves and provide support for the active→model delivery mechanism.

Asynchronous notifications are the primary avenue of communication between active objects and
their models. The notification themselves are instances of class DjinnEvent or its subclasses,
containing information relevant to the particular type of notification. All DjinnEvent objects
carry fields giving the source of the event and a timestamp indicating when it was generated. The
Figure 4.20: Class hierarchy for package djinn.event.
event delivery mechanism is adapted from that used by the Java AWT, extended for distributed operation. Objects which can act as sources of events indicate this by implementing methods with signatures of the form

```java
void add<type>Listener(DjinnListener lis);
void remove<type>Listener(DjinnListener lis);
```

where `<type>` is the name of an event class, for example `ChangeEvent` or `StreamTracker`. Furthermore, the event source creates an instance of `ListenerGroup` for every class of event it can generate. These objects will maintain the lists of subscribers to each event type and deliver the actual events when requested.

Objects wishing to receive events of a particular type must implement the appropriate listener interface, a subclass of `DjinnListener`. The interface will specify a single method

```java
void notify<type>Event(ev);
```

where `<type>` is as above. The object registers as a listener by invoking the `add<type>Listener` method on the desired source, whereupon the appropriate `ListenerGroup` will record the registration.

When a notification is generated, the source object delivers the event to the appropriate `ListenerGroup`, which places it on a queue and returns immediately. In this way, the active object is not held up by the potentially long delays encountered in delivering events to a large group of remote listeners. A background thread in the `ListenerGroup` object arranges for the correct `notify` method to be invoked on each listener. Currently the invocation is made over RMI, using a dedicated thread to ensure that the sender is not blocked.

In the AWT event model, listener interfaces may specify more than one method for each event type; the method called for a particular event delivery serves to further specify the contents of the event. Adding this feature to DJNN would considerably complicate the implementation of the listener group objects, so for the time being event listeners must inspect the fields of the event object to determine the cause of the notification.

**Package `djinn.active`**

This package, unsurprisingly, provides the basic support for the active layers of DJNN applications. The class hierarchy is considerably simpler than the model layer, as shown by Figure 4.21. Application programmers will generally not have to deal with this package directly; component programmers, on the other hand, will be developing new classes that extend the members of `djinn.active`. 
Figure 4.21: Class hierarchy for package djinn.active.
public abstract class ActiveObject extends DjinnObject {
    // Constructor
    public ActiveObject(String name);

    // Methods
    public void deliver(DjinnEvent ev);
    public void notify(DjinnEvent ev);
    public void notify(InvocationEvent ev);
}

Figure 4.22: Class djinn.active.ActiveObject.

Analogous with the ModelObject class in djinn.model, most objects in the active layer of an application are instances of some subclass of ActiveObject (Figure 4.22). This class provides default versions of the state changing methods of Figure 4.9 and support for model→active invocations. In the early prototypes of DJINN, the latter were implemented as direct RMI invocations by a model object on its active counterpart. However, the current framework uses a more indirect method built on the DJINN event mechanisms. To invoke a particular method of an active object, the model invokes, via RMI, the deliver method of the target object, passing it an InvocationEvent that contains the name and parameters of the actual method to be invoked. Then, deliver will look up an appropriate notify method, which—in the case of an InvocationEvent—will invoke the named method with the supplied parameters.

An obvious question at this point is "Why complicate a simple method invocation in this way?" Direct invocations are perfectly adequate when the state of the active object must track that of its model at all times. However, in the course of implementing the reconfiguration mechanisms presented in the following chapters, it became clear that in certain circumstances it is necessary to both defer and reorder the mapping of actions from the model to the active layer. Thus, the indirect event-based technique was developed and, once it was in place, it seemed easier to use it for all model→active invocations rather than support two different mechanisms.

Package djinn.io

The classes and interfaces in this package (see Figure 4.23) provide mechanisms supporting the transport of media and event objects between Djinn runtime environments—across a network connector, for example. In general, to account for the different in-memory layout of objects on different hardware platforms, and to ensure consistency of pointers and other indirect references, objects should be marshalled into a common format for transmission and a new, equivalent object constructed by unmarshalling at the destination machine. The ObjectOutputStream and ObjectInputStream interfaces specify a simple marshalling protocol that is sufficient for all DJINN→DJINN media traffic. These interfaces are implemented by the
ObjectOutputStream and ObjectInputStream classes that convert media objects to and from streams of bytes.

Large objects may need to be fragmented for transmission over a particular communication link; this is the responsibility of the connector object that knows the characteristics of the link. ObjectOutputStream and ObjectInputStream support fragmentation through the BufferOutput and BufferInput interfaces. The output stream will deliver marshalled data in buffers of predetermined size to an object implementing BufferOutput—typically this is the source end of the connector—with the same process occurring in reverse at the destination end. Figure 4.24 shows the objects and data flows involved in transmitting a media object across a network connection.

Package djinn.media

All media data carried by DJINN applications is stored in instances of class djinn.media.Media and its subclasses (Figure 4.25). The internal structure of media classes varies, depending upon the media type and the amount of processing carried out by the Java code, ranging from a completely opaque array of bytes to complex linked object structures. Most media classes will implement the Externalisable interface from djinn.io; the methods in this interface are invoked to request a media object to marshal or unmarshal itself to or from
Network Connector

Figure 4.24: Object marshalling and fragmentation for network transmission.

Package djinn.io

Externalisable

Package djinn.media

Media

Specific media classes

Figure 4.25: Class hierarchy for package djinn.media.
a supplied byte stream. This approach is used, rather than a single generic marshalling engine, because the media classes may be able to produce a more compact marshalled representation: they will know which parts of their structures do not need to be transmitted and how to more efficiently encode certain fields.

4.4 Case studies revisited

At this point I will return to the application case studies first introduced in Section 3.1, and illustrate how these might be implemented as DJINN applications. Leaving aside for the moment questions of QoS management, admission control and the applications' interaction with the rest of the world, this section will concentrate on presenting compositions of DJINN components that implement the desired functionality. The structures shown below are by no means the only—or even the best—realisations of the sample applications; they were chosen in order to demonstrate as many aspects of the architecture described above as possible. Note that the diagrams below show the model components only. The configuration of the active components is identical to that of the atomic (non-composite) model components, except for the physical location of the active components. The following sections include incomplete code snippets giving the interfaces for the different component types.

4.4.1 Case Study 1: Remote Surveillance

The remote surveillance application contains two distinct organisational and administrative domains: the secure site itself, and the networks and equipment operated by the emergency services. Neither organisation wishes to give the other complete access to its internal networks; rather, access is restricted to a well defined interface offered by each domain. The application is therefore structured as two top-level composite components—one for each domain—as shown in Figure 4.26. These two components will be created independently of one another and may in fact be sub-components of larger, site-specific applications run by each organisation.

The internal structure of the SecureSite component is shown in Figure 4.27. The component is modelled as a generic video server component and a collection of clients. The video server consists of a set of video sources—in this case cameras—that receive all of the incoming video streams, optionally (re)compresses them, then delivers the video to any interested clients. The clients may be attached to the local Ethernet or WaveLAN networks, or connected remotely via GSM or the dedicated high-speed link to the emergency services. Clients on the local networks communicate directly with the video server, which has interfaces on both networks. Remote clients, on the other hand, connect to the server through "gateway" components such as the HighSpeedGateway and GSMGateway shown in Figure 4.27. Any of these components, with the possible exception
Figure 4.26: Top-level composite in the remote surveillance application.

Figure 4.27: Structure of the Secure Site component.
class VideoServer extends CompositeComponent {
    // Constructor
    public VideoServer(String name, Camera[] cameras);

    // New methods
    public void addClient(VideoClient client);
    public void removeClient(VideoClient client);

    // Override methods from CompositeComponent
    public void add(Component comp) { }
    public void add(Component[] comps) { }
    public void remove(Component comp) { }
    public void remove(Component[] comps) { }
}

Figure 4.28: Class surveil lanceApp.Video Server.

of the cameras, may be a composite component with further internal structure not shown in the diagram.

As shown, the system is supporting five cameras, three fixed workstations and three mobile terminals. Two of the mobile terminals are monitoring the same camera, and the use of multicast addressing means that only two video streams need to be broadcast onto the local WaveLAN. Two video streams are also being sent over the dedicated link to the emergency services. Note that the component structure in Figure 4.27 gives no indication of the physical connectivity between the cameras, video server and fixed workstations. These components may share a single Ethernet segment, or there may be separate segments for the cameras and workstations with the server attached to both.

The video server is implemented by an instance of class VideoServer, an application-specific subclass of CompositeComponent (Figure 4.28). VideoServer overrides the add and remove methods from CompositeComponent in order to have complete control over its own internal structure through the addClient and removeClient methods. The client objects passed to these methods may be physically located on Ethernet, WaveLAN or GSM clients. The constructor is given a set of Camera objects, which will be encapsulated within the composite component and offered to clients.

Figure 4.29 shows the internal structure of the EmergencyServices composite component. This component is superficially similar to the SecureSite component, except that the video streams arrive via the dedicated link to the secure site rather than directly from cameras. There is also the possibility of additional streams sourced from within the emergency services' application, although these are not shown here. The system is shown supporting two incoming streams from the secure site, which are being re-broadcast to three fixed workstations and two mobile terminals
connected over GSM links. As with the SecureSite, these components may be composite components with additional internal structure.

4.4.2 Case Study 2: Digital Television Production

The digital television editing system presented in Section 3.1.2 is really three applications in one:

1. Format conversion and browse track generation, transforming source streams into a format suitable for the editing system and storing them on the server.

2. The editing workstation, where a human operator composes audio and video with various effects and transitions to produce a description of the finished programme. The output of this application is an edit decision list (EDL).

3. Finally, the edit-conforming switch itself. This takes the stored streams and EDL produced by the previous two stages and produces the final broadcast-ready output stream.

Figures 4.30–4.32 show the top-level components that might make up DJINN implementations of each of these applications. It is worth noting that these three subsystems must be run in sequence for the production of any given programme and that the time intervals between each stage may be arbitrarily long, since the outputs of the first two stages can be stored until they are needed. On the other hand, the three applications can be used in parallel to work on different programmes simultaneously.
Figure 4.30: Format converter/browse track generator application.

Figure 4.31: Editing workstation application.
The browse track generation and editing workstation subsystems are not particularly interesting from DJINN's point of view: they are both just simple processing pipelines whose structure is unlikely to change and without any significant real-time requirements (although it is obviously desirable for them to complete their work as quickly as possible). For this reason, I will concentrate on the more complex and dynamic edit-conforming switch subsystem.

Figure 4.33 shows the internal structure of the switch in more detail. Note that the EDL is read through a port attached to the composite component. The source of EDL data is determined by the controlling application: for example, it could be a file or real-time output from an editing workstation. The EDL input port does not map to a port on any atomic component, nor does it have a physical representation in the active layer. Indeed, the component could have been built to read EDL data from files whose names were passed in via a method invocation. However, making use of the port mechanism in this way is both in keeping with the DJINN design philosophy and supports a more flexible style of application construction.

An instance of the EditConformingSwitch class is a long-lived component: it is intimately bound to the underlying switching hardware—over which it has exclusive control—and should not need to be shut down unless the hardware is changed or the component software updated. Thus, the component will be powered on almost as soon as it is created and not powered off again until it is ready to be deleted.

Figure 4.34 gives a simple skeleton implementation of the EditConformingSwitch class. When it is first created, the component is passed a list of references to servers storing the broadcast-quality audio and video clips that will make up the output programmes. If this list is not static, it may be more efficient for the component to discover server locations dynamically through some kind of directory service. Note that the application does not need to know where its output stream(s) will be routed to: this is a decision to be made at a higher level by whatever user or software is controlling
Figure 4.33: Internal structure of the edit-conforming switch.
The DJINN Architecture

```java
class EditConformingSwitch extends CompositeComponent {
    // Constructor
    public EditConformingSwitch(String name, VideoServer[] servers);
}
```

Figure 4.34: Class DTVAEdit.ConformingSwitch.

the editing application.

The most interesting behaviour of the editing system, namely its dynamic reconfiguration in response to the instructions encoded in the EDL data, will be presented in detail in Chapters 5 and 6.

4.5 Benefits and Problems

While a full evaluation of the DJINN architecture will be deferred until Chapter 7, I feel it is appropriate at this point to discuss some of the perceived advantages and drawbacks of the framework, particularly with regard to some of its more novel features.

The most obvious benefit accruing from the use of the application model is the separation of concerns achieved between the logical and physical structure of an application. Physical structure is entirely described by the configuration of active components: their locations, interconnections and internal settings reflect the actual hosts, multimedia hardware and network links in the distributed system. Furthermore, each active component operates essentially independently of its fellows, carrying out its particular processing task without any knowledge of where the data comes from, where it goes to or how the component contributes to the application as a whole.

On the other hand, the model layer of an application describes—amongst other things—the logical structure of the system. Composition allows a set of components working towards a common goal to be encapsulated within a well-defined interface\(^3\) representing some higher-level function carried out by the application, such as security monitoring. The same set of components could equally well perform the video-conferencing function if enclosed within a different composite component shell. At this level, the behaviour of the composite component is far more important than the exact arrangement of active components that ultimately implement that behaviour. Composition eases the construction of robust federated systems by restricting the interactions between federation members to those permitted by the interfaces that each member exports.

In addition, the model serves as a repository for what can be thought of as application "meta-data", such as QoS and reconfiguration characteristics that cannot, in general, be associated with any one

---

\(^3\)In this context, the term "interface" refers to the combination of media data ports, method invocations and event I/O supported by a particular component class.
active component. In the case of QoS properties, while most of these are specific to a particular component running on a particular host, all but the simplest application will have inter-component QoS dependencies that are most conveniently stored by a composite component in the model layer. Likewise, any non-trivial reconfiguration will involve multiple active components. None of these components have any way of determining how a change to their configuration will affect the rest of the system. DJJNN's modelling and composition mechanisms allow programmers to build up complex distributed applications without needing to understand, in detail, the behaviour of every atomic component in the system.

The availability of generic model components that can map onto many different physical combinations of active components leads to applications that are naturally portable and decoupled from the underlying computing environment, such as a video-conferencing application that can be instantiated anywhere that a camera, microphone, display, speaker and network interface(s) exist, regardless of the exact hardware and operating system involved.

However, the DJJNN approach is not without its drawbacks. One difficulty that has become apparent during the development of sample applications is the additional work required to produce a new atomic component class. As well as the active component code, which may need to be developed separately for each hardware/OS platform to be supported by the component, the programmer must also create the model class for the component. For most atomic components the model code is not especially complex, but the work is repetitive and error-prone, and potentially amenable to automation in the form of a tool to generate skeleton code for the model class. The problem is exacerbated by the fact that—beyond the required powerOn, powerOff, start and stop methods—the model-to-active interface for atomic components is not restricted in any way. This issue is of particular concern when a single model component must interact with several subtly different active components on different platforms.

Although the hiding of low-level detail offered by the composition mechanism is useful when building complex applications, there is a danger that some item important to a particular application may be hidden by an intervening composite component. For instance, a specialised application might wish to use a unique feature of a certain camera; unfortunately, the generic Camera model component does not expose this feature to higher levels of the component hierarchy even though the active component supports it. This is of course a general problem with any mechanism for enforcing encapsulation, data hiding or modularity. The most feasible solution, other than violating component encapsulation to access active component features directly, is to create a subclass of the generic model component that exports the desired items. An unfortunate side effect is the likely proliferation of such subclasses for each minor variant of the active component, negating the benefit of having generic model components in the first place.

Finally, it must be said that the use of Java as the main implementation language has had a noticeable impact on DJJNN's performance. Although it is difficult to make quantitative comparisons,
the frame rate achieved by a DJINN-based MPEG decoder, for example, is often 30–50% less than a native decoder on the same platform. Similarly, a DJINN application will generally use significantly more resources (CPU bandwidth, memory) to achieve the same level of service as an equivalent native program. This is despite many of the most computationally intensive parts of the DJINN runtime—video codecs and marshalling of bulk media data, for instance—being implemented as native code libraries. It is possible that recent advances in Java optimisation such as Sun’s HotSpot JIT compiler and runtime optimiser may help to reduce this performance gap.

4.6 Summary

This chapter has presented the architecture of DJINN, a programming framework for distributed, real-time continuous-media applications. DJINN implements a layer of middleware “glue” between applications and operating systems, encapsulating and enhancing low-level services to provide a platform supporting a wide range of multimedia applications. As well as a collection of APIs and associated class libraries, DJINN offers a runtime system that includes services for dynamic application control, QoS management and reconfiguration.

Applications are built under a two-tier component architecture. Programmers view an application as a network of autonomous interconnected components, where each component may be a composite component: an encapsulation of simpler components with its own interfaces and unique behaviours. The framework maps this logical model of an application onto the lower tier of active components corresponding to the actual hardware and software entities needed to perform the application’s function. The main advantages achieved through this model-based approach are firstly the separation of concerns between the logical structure of an application and the physical configuration of media-processing components that realise it; and secondly the capacity of the model to store application “meta-data” that cannot be expressed effectively by the active components, such as complex reconfiguration behaviour distributed across multiple components, or QoS characteristics and inter-component constraints.

This chapter has primarily addressed the static aspects of DJINN, using application modelling techniques to construct a single, stable configuration of components. Chapters 5 and 6 will discuss dynamic reconfiguration of DJINN applications, in particular the algorithms and programming abstractions I have developed to support maintenance of QoS during reconfigurations.
CHAPTER 5

HIGH-LEVEL RECONFIGURATION
ARCHITECTURE

This chapter presents a high-level approach to the reconfiguration of distributed multimedia systems. Reconfiguration of these soft real-time systems imposes some strong constraints on the reconfiguration process, as presented in Chapter 3. These requirements arise primarily from the observation that continuous-media systems must in general—by their very nature—remain fully operational during reconfiguration activity. Any errors or failures produced by the reconfiguration process cannot be transparently rolled back or recovered from, as they are probably already visible to users of the system as unwanted output artifacts. My solution to this problem is, in the most abstract view, centred around the two-level model-based approach to application structuring described in Chapter 4. Reconfigurations are performed firstly on the model objects of an application, with the configuration changes then analysed for both structural integrity and acceptability to the QoS system before being applied to the corresponding active object layer. This chapter presents the first half of this procedure—essentially the operations on the model object layer only, while Chapter 6 discusses how the changes to the application model are mapped onto the active layer. As far as the model layer is concerned, the requirements from Chapter 3 are met through the structuring of reconfigurations as atomic actions; an adaptation of conventional transaction-based mechanisms for ensuring consistent updates to a (non real-time) data store. The remainder of the chapter is organised as follows:

- Section 5.1 provides a definition of reconfiguration within the two-layer application model and discusses the implications of this definition with regard to ensuring application consistency. The definition of reconfiguration is abstracted from sample reconfigurations on the two case study applications, leading to a more formal definition that can be used in the development of reconfiguration mechanisms.
Section 5.2 describes the high-level configuration architecture itself. This section motivates the idea of atomic actions as a structuring mechanism for reconfiguration and presents the fundamental properties and semantics of atomic actions.

Section 5.3 discusses the prototype implementation of this architecture used by the DIJNN framework.

Section 5.4 shows how the atomic action architecture might be used to add reconfiguration capabilities to the sample applications introduced in Chapter 3.

5.1 Defining Reconfiguration

An important early step in developing the reconfiguration architecture is to define exactly what is meant by the term “reconfiguration”, in the context of an application structured according to the two-layer model of Chapter 4. Like most other interactions with a DIJNN application, reconfigurations are always performed on the model layer of the application—the active object layer can then be updated to match the new configuration of the corresponding atomic objects in the model layer. The rationale for this approach is twofold: firstly, if unconstrained changes to the active layer are permitted, without reference to the model layer, the relationship between the layers breaks down. It is generally impractical, and often impossible, to update the state of a composite component to accurately reflect the state of its underlying atomic components, whereas the reverse operation, that of “pushing” composite component state downwards towards the leaves of the application’s component tree, is relatively trivial. Secondly, by updating the model layer first, new configurations can be evaluated against an application-specific metric that determines whether or not the active layer should be updated. Unsuitable reconfigurations are thus prevented from causing damage to the live application.

This view of reconfiguration does not preclude the situation where a reconfiguration is initiated from the active layer of the application, say by an active object sending a notification to its model counterpart to indicate that some exceptional event has occurred. However, the active object in question cannot alter its own configuration until it is instructed to do so by the model layer.

It is useful to first examine some “typical” reconfigurations of real applications, as a foundation for the more abstract and formal definition presented below. Consider the following example reconfigurations, performed on the two case study applications of Chapter 3:

5.1.1 Case study 1: Remote Surveillance System

Some of the typical—and frequently occurring—reconfigurations that may be performed on the remote surveillance system are:
A user selects a new video stream to view. The user may be using a mobile terminal or a fixed workstation. In the mobile case, limited network bandwidth and processing capability on the terminal mean that only a single stream can be received and played out at a time. Thus, any stream that the mobile user was already viewing must be shut down. On the other hand, the more powerful fixed workstations are in general able to receive and play up to $N$ streams simultaneously; if the addition of the new stream would exceed this limit then either the new stream should be rejected or an existing stream shut down.

A mobile user moves out of the range of the WaveLAN network. The user's terminal should be reconfigured to use a low-bandwidth GSM connection, with a corresponding reduction in the quality of the video that is received. The reverse reconfiguration is also supported, with the user moving back into the range of the WaveLAN and receiving high-quality video once again. In both cases, the reconfiguration should be seamless; that is, the user should not experience any unexpected changes in perceived quality of service due to the reconfiguration. In this application, the change in video frame rate and image quality experienced when switching between networks is entirely expected and must not be considered a violation of reconfiguration constraints. Any other observable artifacts of the reconfiguration—such as a long pause in playback or an obvious jump in the playback position of the stream—are unexpected and should be avoided if possible. Either of these reconfigurations should be rejected if there is insufficient bandwidth or other resource to support a new connection to the target network.

For any of these reconfigurations, users on other workstations or terminals should not be affected by the reconfiguration, even if they are viewing the same stream(s) as the reconfiguring user. That is, other users should not be able to observe a reduction in their perceived quality of service as the reconfiguration takes place.

Throughout the rest of this chapter, I will examine the particular reconfiguration case of a mobile user moving from the WaveLAN network to a GSM connection in greater detail. Figures 5.1 and 5.2 show the "before" and "after" states of the surveillance application undergoing this reconfiguration.

As the diagrams illustrate, the most obvious effect of this reconfiguration is the GSM connection that is created between the main video server and the mobile client. The MPEG decompression component on the client is shut down and replaced by a H.263 decoder. Note that the configuration of the video server remains essentially unchanged, except that one of its multicast clients is now the GSM gateway rather than the remote node itself. Prior to the reconfiguration, no GSM-connected clients were viewing this source stream, so a new H.263 compression component must be instantiated in the GSM gateway to deliver the frames in the required format. If a second GSM client was to request the same stream, the gateway would arrange for it to be multicast to both clients, across two separate GSM connections.
Figure 5.1: Remote surveillance application before reconfiguration.

Figure 5.2: Remote surveillance application after reconfiguration.
5.1.2 Case study 2: Digital Television Production

The MPEG editing application described previously is only one part of a larger digital television studio system: this is a significantly more complex application than the remote surveillance system and supports a correspondingly wider variety of reconfigurations. Most, however, are variations on the basic theme of re-routing streams from source components through different networks of transforming and filtering components, to their eventual destination(s). Some typical examples include:

- Switching between multiple cameras in a live studio session.

- Mixing input from multiple audio sources into a single stream, synchronised with the video stream for a programme. The set of input streams contributing to the output can be changed dynamically, in real time.

- Switching the broadcast output between "programme" and "commercial" streams.

- Splicing together a sequence of independent video clips to form the broadcast output during a commercial break.

- Superposition of computer-generated imagery—which may be rendered in real-time or pre-recorded—over live output from, for example, the news studio.

- Online, collaborative editing of a programme, composed from multiple pre-recorded audio and video streams.

Likewise, within the editing application, reconfigurations will involve the creation and shutdown of input streams to the edit conforming switch (see Section 3.1.2), and re-routing of those stream through the various switching and image processing components within the switch. Figures 5.3 and 5.4 show such a reconfiguration. The switch has just performed a cut between two input streams; after the reconfiguration a third stream—perhaps containing computer-generated material—has been mixed in and overlaid on the output. The diagrams show the video streams only; switching and processing of audio streams proceeds in parallel with the video.

These activities are all commonplace in today's existing digital studios. However, the more complex tasks such as programme editing and the composition of computer-generated and real-world imagery are usually not performed in real-time due to the limited processing capacity of current equipment. I believe that future systems will not suffer from this limitation, allowing most editing and processing tasks to be carried out in real-time (It should be noted that limitations on bandwidth are unlikely to be an issue for this application, even today). The most important considerations, especially when dealing with live streams or broadcast output, are ensuring seamless stream switching
Figure 5.3: Edit-conforming switch before reconfiguration.
Figure 5.4: Edit-conforming switch after reconfiguration.
and maintaining synchronisation between multiple streams undergoing reconfiguration. The quality of service requirements for a television programme broadcast are considerably higher than those for the surveillance application so the constraints on reconfigurations are correspondingly much tighter.

The components and paths in the studio system—and the streams they process—can be arranged into a hierarchy that determines their priority in terms of access to resources and hence their ability to be reconfigured to use those resources. If the studio is engaged in both broadcasting and recording programmes for later broadcast, then clearly the components and streams producing the current, active broadcast stream should be given the highest priority, as this stream must be played out in real-time. That is, the components involved in the broadcast should never have to compete for resources and nor have their quality of service level reduced to accommodate other activities on the system. Paths processing live streams not destined for immediate broadcast output will have a somewhat lower priority than the broadcast components, since these streams are important (there is usually only one chance to record a live event) but nevertheless should not be allowed to interfere with the broadcast. At lower priorities still are paths that deal only with pre-recorded or computer generated streams. These paths do not have to run in real-time and can be stalled indefinitely until resources are available.

As with the surveillance system, reconfigurations of the television studio applications should not have any effect on streams, components or users unconnected with the particular reconfiguration. In the television studio, this requirement must be extended to account for the priority scheme discussed above: reconfigurations of high-priority paths will be permitted to override the existing configurations of lower-priority paths, even to the extent of “stealing” resources and lowering the quality of service of other paths.

5.1.3 A more precise definition of reconfiguration

The above examples demonstrate that a reconfiguration transforms the structure and/or quality of service properties of an application in a well-defined way, while maintaining certain constraints on both the reconfigured components and the application as a whole. Reconfiguration is often defined informally (and rather obviously) as any change in the configuration of a system, with the term *dynamic reconfiguration* used for changes that occur while the system is executing. As a starting point for a more formal and precise definition of reconfiguration, relevant to the application domain addressed by this research, consider the following offered by Kramer and Magee in [KM90] amongst the objectives of a dynamic “change management” system:

“... A system is viewed as moving from one consistent state to the next. ... In order to avoid the loss of application transactions and achieve a consistent state after change, a consistent application state is required in the affected part of the system before the change.”
That is:

A reconfiguration encapsulates a transition of the application between two consistent states, the initial and final configurations.

This definition raises two further questions:

1. What constitutes a consistent state?
2. How is the transition from the initial to final configuration achieved?

5.1.4 Consistent state

Kramer and Magee define a consistent application state [KM90] as:

"...one from which the system can continue processing normally rather than progressing towards an error state. It is usually expressed in terms of some global system invariant."

Thus, the notion of a consistent state is largely application-specific. For example, a video-conferencing system might define a consistent state as one in which all participants are receiving audio and video streams from the same single participant, that being whichever participant currently “has the floor” under the application’s floor-control régime. On the other hand, the remote surveillance system would probably use a very different definition of consistency, to capture the fact that users are viewing many different streams and that certain users may take priority over others.

However, despite the generally application-specific nature of a consistent state, a number of generic properties can be defined that are common to all consistent states and can be applied at the level of individual components and ports. In particular, the following are necessary (although not always sufficient) conditions for an application to have a consistent configuration; that is, they form part of the global consistency invariant for the application in question:

- The application has passed an admission control test and a sufficient quantity of resources have been reserved to meet its stated QoS requirements. Note that these are not necessarily “hard” resource reservations for particular fixed quantities of certain resources. If the application is able to be flexible about its requirements then any resource allocation between its minimum and maximum requirements will be sufficient to satisfy this property.

- The inverse of the previous property must also hold in order to have a consistent state: the application must guarantee not to use more resources than have been reserved for it.
• All of the paths in the application are fully connected; that is, there is a data path between the source and sink nodes of each path. This does not imply that the same data is consumed at the sink of the path as was produced at the source, since the path may include arbitrary filtering and transforming components. It is sufficient that there is a "depends-on" relationship between the source and sink nodes, meaning that there is a mapping between individual data elements at the source and sink.

• All running ports—those actually producing or consuming data on behalf of their components—are connected to another port of the appropriate direction and media data type. There is no requirement for inactive ports to be connected to anything.

All of the above conditions are applicable to every component and port in an application, irrespective of the high-level structure of the system or the exact class of each component and port. Atomic port and component subclasses, as well as composite component classes, are likely to have their own class-specific consistency constraints. For example:

• An atomic component such as a camera might check that suitable camera hardware exists on its active component host before attempting to create the active component.

• Composite components should verify that any port interconnections crossing the composite component boundary are correctly routed through proxy ports. That is, a port of an encapsulated component cannot be directly connected to a port on an outside component.

• More complex composite components are likely to perform additional consistency checks on their internal structure, such as the video conference example given above. As a further example, a video server component can service a certain number of clients given the resources supplied to it. Additional client connections should be refused once this number is reached, as adding more clients would result in an inconsistent state where the component used more resources than were allocated to it.

In general, then, every model object class \( M \) defines a set of parameterised consistency constraints. Furthermore, each superclass of \( M \) defines its own set of less-specific consistency constraints that will also apply to objects of class \( M \). If \( A \) is an application constructed from the \( N \) distinct model objects \( O = \{O_1, \ldots, O_N\} \), then \( A \) is consistent iff every element \( O_i \) of \( O \) is consistent. When \( O_i \) is an atomic component of class \( M_i \), it is consistent iff it meets all of the constraints declared by \( M_i \) and each of its superclasses. Where \( O_i \) is a composite component, the definition is applied recursively to the subcomponents of \( O_i \). The composite component class may of course declare its own class-specific constraints that will also be validated.

The important thing to observe about all of these consistency properties, whether application-specific or otherwise, is that they are essentially static properties. A certain application-defined set of
properties must hold in order to have a consistent state, but once this is known to be true it follows that it remains true until the application is reconfigured again. The idea of consistent states can thus be likened to the use of pre- and post-conditions on a function in a program, with the transition between the two states analogous to the processing carried out by the function. In addition, many of the consistency properties above can trivially be made invariant, guaranteed always to be true. This eases the burden of verifying an application state as consistent by removing the need to check invariant properties.

5.1.5 Performing the transition

At the most fundamental level, the transition between the initial and final configurations of an application is described by the following two-step process:

1. Make all of the required changes to the application state.

2. Check that the new state is consistent, according to the application’s definition of consistency. If it is consistent, activate the new configuration. If not, return to the previous consistent state.

In the present case, where the application is structured according to the two-layer model used by DIJNN, these steps are performed on the model object layer only and a third step must be added to the list:

3. Update the active object layer to match the new configuration of the application model.

Unfortunately this simple scenario is complicated by several factors. Firstly, the application is required to return to its initial configuration if it turns out that the final configuration is inconsistent. This is simply a restatement of the atomicity requirement from Section 3.4. A mechanism must be provided to achieve this without forcing the user or programmer to explicitly undo or reverse every configuration change that was made. Secondly, it is preferable that observers of the system—users or programs not directly involved in performing the reconfiguration in question—do not see all of the inconsistent intermediate stages that the application passes through between the initial and final configuration. In particular these observers should not be aware that a reconfiguration was started and then reversed due to an inconsistent final state. Thirdly, the final step of updating the active object layer must be given special consideration, since it is manipulating continuous media streams that are probably active while the reconfiguration is taking place.

This third point leads into the idea of dynamic consistency as opposed to the merely static consistency considered above. If it was necessary to ensure only static consistency during a reconfiguration, the entire active layer of the application could be shut down before making any changes to its configuration and restarted once all of the necessary updates had been applied. Such a procedure would
ensure that the active as well as the model layer was moved atomically from the initial to the final configuration, both of which are known to be statically consistent. However, unless this shutdown-modify-restart sequence can be performed quickly enough that it has no observable effect, this naive technique is unlikely to prove satisfactory. For example, in the remote surveillance application it is clearly unacceptable—not to mention potentially dangerous—to deny service to all the users of a particular video stream during a reconfiguration to add a new recipient of that stream. I believe that in general, and insofar as this is possible, users will want their applications to continue running uninterrupted during reconfiguration, and for the reconfiguration not produce any unwanted output artifacts. Thus, the definition of a reconfiguration must be extended to encompass the maintenance of dynamic as well as static consistency:

"... It should not be necessary to stop the whole of a running application system to modify part of it. ... The rest of the system should be able to continue its execution normally."

[KM90]

So the definition becomes:

A reconfiguration encapsulates the transition of the application between two statically consistent states and maintains the dynamic consistency of the application for the duration of the reconfiguration.

Clearly, it is unreasonable to apply the static consistency criteria enumerated above as measures of dynamic consistency, for the simple reason that most of the states an application passes through in the transition between its initial and final configurations will be inconsistent under those criteria. To ensure that dynamic consistency is maintained in the active object layer of an application undergoing reconfiguration, the following conditions must be met:

1. The reconfiguration must be performed atomically. That is, unless it is certain that all of the changes to active objects related to a particular reconfiguration will succeed, none of the changes should be made. This is analogous to the atomicity requirement for changes to the model object layer, except that it is not generally possible to reverse changes to active objects. Thus, no changes can be made until it is known that the whole reconfiguration will succeed.

2. No garbled, corrupted or otherwise erroneous data should be produced by the actions of the reconfiguration. For example, the internal buffers of a MPEG decoder must be flushed before it begins decoding a new stream, otherwise corrupted frames will be generated. I will refer to this specific requirement as data consistency.

3. Likewise, the temporal properties of streams affected by the reconfiguration should not be violated. For example, there should not be a long pause or dramatic change in frame rate
when switching a video player from one input stream to another, unless these are called for by the reconfiguration. This requirement will be referred to as temporal consistency.

4. Objects not directly participating in the reconfiguration are not affected. That is, unless the configuration of a given active object is actually being changed, that object should not be aware that a reconfiguration has taken place.

Note that the requirement for temporal consistency in particular is not intended to affect the range of reconfigurations that can be performed. It is perfectly acceptable for the reconfiguration itself to result in large changes to the temporal parameters of streams (frame rate, latency, etc.). What is unacceptable is for such changes to be caused by the intermediate stages of the transition to the final configuration.

Dynamic consistency is the subject of the of the next chapter. It turns out that it is not possible in a resource-limited environment to absolutely guarantee that all of the above consistency conditions will be met. However, a principled, algorithmic approach to the problem allows both a determination of how much divergence from the ideal situation is necessary, and the ability to then make a reasoned tradeoff amongst the conflicting claims of resource usage, consistency and timeliness.

The remainder of this chapter discusses the use of atomic actions to manage atomic, statically consistent reconfigurations to an application's model object layer.

5.2 Atomic actions

From the discussion above it has emerged that the major requirements for reconfiguration, from the point of view of the model object layer, are atomicity and consistency. Along with isolation and durability these are two of the main benefits provided by the use of transactions [HR83, AA92, GR93, CDK94e] in traditional database processing applications. This overlap of requirements has led to my consideration of transactions as a possible mechanism for managing the reconfiguration of component-based multimedia systems. If the model object layer only is considered, the scenarios are superficially similar—both involve the transformation of a collection of "objects" (database records versus application model objects) from one consistent state to another. Given this motivation, I have adapted the semantics of traditional database transactions to the application domains addressed by the DIIINN framework. Reconfigurations are encapsulated within transactional entities that maintain the properties discussed above, on behalf of the application. The transactional entities themselves were originally referred to as multimedia transactions but have more recently been renamed as atomic actions in order to avoid the impression that they support a complete extended transaction architecture [ELMB92, BÖH92, BÖ97, MRJ97]. The atomic action mechanisms presented here do not fully implement all of the conventional properties of transactions, although it would be relatively trivial to do so. For the most part, however, atomic actions do
behave in fundamentally the same manner as database transactions. In particular, they support the isolation property and can be aborted and rolled back to the previous consistent configuration.

5.2.1 Assumptions and design goals

Consider the following two basic operating scenarios for transactional reconfiguration of a multimedia system, each with its own implications for the choice of concurrency control and conflict resolution algorithms:

Interactive reconfiguration

In the first case, reconfigurations are unstructured, long-lived interactive operations on an application model, driven by a human user. Such reconfigurations could be performed on a model representing a running application or on an inactive model under development. In either case, the model might be shared by multiple users who are concurrently reconfiguring it. Since the reconfigurations are long-lived—possibly on the scale of days or even weeks—users must face the possibility that their changes may not ultimately be allowed to commit, if another user has already committed a conflicting update. This scenario implies the use of an optimistic concurrency control scheme, with multiple tentative versions of components in existence [KR81, CDK94a]. Alternatively, the system could rely on “social protocols” to maintain consistency; the GroupKit project [RG96] has shown that this approach will work in certain situations.

Program-driven reconfiguration

In the second mode of operation, users have no control over the execution of reconfigurations except to initiate them. These reconfigurations are short-lived, predefined operations carried out directly by programs. They might be carried out against a potentially shared application model as in the previous scenario, or on the actual running application. However, their shorter lifetimes and more deterministic behaviour mean that a simpler pessimistic concurrency control scheme [CDK94a] will be sufficient to maintain application consistency.

The types of operations that would be carried out in each style of reconfiguration are also quite different. In the interactive case, the likely scenario is that new component classes—or a whole new application—are being developed. If the user is developing a new atomic component that models some physical multimedia hardware, then eventually the corresponding active component classes for the target platform(s) will need to be instrumented and tested to derive QoS data for the model. However, the remainder of the new component’s behaviour is entirely represented by the model, so it can be prototyped and tested extensively without any involvement from the active object layer. This particular benefit of the two-layer approach is even more marked in the case of composite
components, whose activity is almost entirely confined to the model layer. In this mode of working users are likely to be rapidly modifying existing classes and testing application and reconfiguration logic more than the component's real-time behaviour.

In the program-driven mode, the application will generally have been prototyped, tested and built, and is now running in a production setting. The reconfigurations in this case are pre-defined sequences of actions developed during the prototyping phase above. Reconfigurations are run in response to user interaction or other external stimuli; there is no manual user or programmer control of the reconfiguration. These program-driven reconfigurations will in general involve both the model and active object layers.

Both reconfiguration models could exist in parallel; indeed, this is a long-term goal for the DJINN framework. The interactive mode would be used to prototype and test new reconfiguration operations, or for one-off ad-hoc changes to a running system. A freshly designed operation could then be converted into a short-lived reconfiguration program that would be run against active applications. I view interactive reconfigurations as operating almost exclusively on the model object layer whereas programmed reconfigurations modify the model layer only with the intent of also updating the underlying active objects, thus altering the flow of media data. Whilst it is hoped that DJINN will eventually support both styles of reconfiguration, this dissertation deals only with the program-driven variety. Given this restriction, I have made the following basic assumptions about program-driven reconfigurations on a distributed multimedia application. These have significant consequences for the design of the atomic action architecture, in particular the algorithms used for concurrency control and conflict resolution:

- Reconfigurations are short-lived, well-defined, parameterised operations performed by programs in response to user input or other external stimuli (such as a timer, an event received from an active layer object, in reaction to resource shortages, etc). "Short-lived" is taken to mean "on the order of a few seconds".

- Reconfigurations occur relatively frequently, on time scales from the order of seconds upwards.

- Minor reconfigurations, such as QoS adjustments or floor control in a conferencing system, will be far more frequent than major structural reconfigurations such as adding a participant to the conference.

- The most likely source of conflict between reconfigurations will be concurrent attempts to reconfigure the same high-level composite component. In many cases, however, the sets of lower-level components actually being changed by these reconfigurations will not in fact overlap. This means that, provided the composite component in question correctly manages concurrent accesses to its own internal data structures, such reconfigurations can be run in parallel without resulting in an inconsistent final configuration.
Given these assumptions about the nature of reconfigurations on a distributed multimedia system, we can produce a set of design goals for our atomic action architecture. Because the atomic action mechanism is just one part of the overall DJINN framework, it necessarily reflects some of the decisions and biases of DJINN's design, as well as the requirements derived for configurable distributed multimedia system in general. However, for the most part these design goals should be applicable to other multimedia frameworks operating under similar sets of assumptions:

- Atomic actions should encompass all types of reconfiguration actions: structural changes such as adding components or altering inter-component connections, QoS changes and actions that fit into neither of those categories, such as changing the aperture and shutter speed settings on a networked camera.
- Atomic actions with overlapping scopes but non-conflicting effects should be able to run in parallel and commit successfully.
- Nested atomic actions [WS92] should be supported, to allow greater flexibility and robustness in the structuring of reconfigurations.
- The use of atomic actions should not impose an undue burden on either application users or component programmers.
- Atomic actions should be executed efficiently.

5.2.2 Properties of atomic actions

In [HR83] Härder and Reuter present the "ACID" properties of traditional database transactions:

Atomicity. An atomic action must be "all-or-nothing". That is, either all the actions of the reconfiguration are successfully completed, or none of them are. In the case of continuous media systems, the implication of this property is that no active objects can be modified in any way until it is absolutely certain that all of the changes made by the reconfiguration will succeed. This is because any changes made to active objects will cause observable changes in the behaviour of the application—an incorrect reconfiguration may lead to undefined behaviour or even an application failure. While the changes made by the reconfiguration can often be reversed it is not possible to take back any data produced by an incorrectly configured active layer.

Consistency. An atomic action takes the system from one consistent state to another consistent state. The definition of a consistent state will depend on the nature of the data and the application. The definition of consistent is that developed in Section 5.1.4 above, encompassing both static and dynamic consistency. Note that this definition of the consistency property
Atomic actions

does not say anything about the state of the model layer during the transition between the initial and final consistent states—it will almost certainly pass through a number of statically inconsistent states. Likewise, the usual correspondence between the model and active layers is not necessarily maintained during the transition—the active layer must maintain dynamic consistency regardless of the current state of the model.

Isolation. Each transaction should be performed without interference from other transactions. That is, the intermediate and possibly inconsistent states of a transaction should not be visible to other transactions. This implies that there exists a mechanism for assigning reconfiguration operations to a particular atomic action, or that there is the notion of a "current atomic action" that all operations are implicitly assigned to.

Durability. The effects of a successfully committed transaction should be persistent, even in the presence of process and machine crashes. That is, the effects of the transaction should be written to permanent storage before making them visible.

In the case of atomic actions on multimedia systems, the atomicity and consistency properties are clearly applicable, as these are the two primary motivations for using a transactional mechanism in the first place. In an architecture supporting long-lived, interactive transactions it may be useful to weaken the isolation property somewhat, allowing users to see what others are doing and perhaps even participate in each others' reconfigurations [BK91]. Such a scheme can work well in a situation where human users are on hand to resolve conflicts, but will be problematic in a system where the reconfigurations are entirely under program control. Thus, atomic actions in my architecture also maintain the isolation property.

The durability property is somewhat orthogonal to the design goals stated above. It has its roots in the world of online transaction processing systems where all data is kept persistently in storage media that can survive a server crash. DITNN objects are not currently persistent, but could be made so relatively easily using (for example) the Java serialisation mechanism or an extension to the Java Virtual Machine such as P[ama [ADF96]. At that point it would be useful to ensure that atomic actions maintain the durability property, but at present the system operates under the rather weak assumption that DITNN processes do not crash or lose data.

Atomic action semantics

The basic semantics of an atomic action are identical to those of a traditional database transaction: a program initiates a new atomic action \( A \), performs a sequence of operations within the scope of \( A \), then commits the atomic action, making the changes permanent and visible to entities outside the scope of \( A \). The atomic action may also be aborted—either synchronously by the atomic action program or one of the objects it operates on, or asynchronously by the atomic action support sub-system if it determines that the atomic action cannot be committed for some reason. In particular,
an atomic action should be aborted if it would leave the system in an inconsistent state. When an
atomic action is aborted, any changes it has made to the system will be rolled back, leaving the sys-
tem in a state as though the atomic action had never been initiated. This is not necessarily the same
state that the system had before the atomic action was initiated, as other more successful atomic ac-
tions may be executing concurrently with the aborted atomic action. The scope of an atomic action
may be broadly defined as the set of entities participating in the atomic action. In the case of an
object-oriented system, this definition covers any object where the atomic action reads or writes a
field, or invokes a method, either directly or through another object within the scope of the atomic
action. Entities outside the scope of a particular atomic action are not aware of the changes it has
made to the system until after the atomic action has successfully committed, at which point all of the
changes become visible atomically; that is, entities outside the atomic action see a single, effectively
instantaneous transition between the initial and final states of the system.

The very specific nature of the applications supported by atomic actions means that there is some
deviation from the basic database transaction model. The most significant differences lie in the
precise semantics of the commit operation, which must be modified to update the state of both the
model and active layers of an application, while maintaining the dynamic consistency of the active
layer. The details of how this is achieved will be presented in the next chapter. Table 5.1 lists other
relevant additional properties and semantics of atomic actions as implemented by DJINN. It is
important to note that this is by no means the only, or the best, way that atomic actions could have
been implemented, but more an artifact of DJINN's evolution and the properties of the underlying
Java environment. However, the implementation does fully support the basic ACID properties,
excepting durability, and the notion of consistent reconfiguration developed above.

5.3 Atomic action implementation

It is worthwhile to briefly examine the current implementation of atomic actions within Djinn, to
assist understanding of the sample reconfiguration code presented later and to illustrate the diffi-
culty inherent in implementing system-level functionality within the virtual machine environment
provided by Java. Atomic action support for DJINN applications is provided by the five Java classes
in the package djinn.transaction, along with a shared library of C code for each architec-
ture that DJINN runs on:

5.3.1 Package overview

Class TransactionManager

Atomic actions are considered to be local to a thread of execution in the application; that is, each
thread in a DJINN application has its own "current" atomic action, which will be null if there is
• Atomic actions are performed on the model object layer of an application. The configuration of the underlying active object layer will be updated to match the new configuration of the model layer iff the atomic action successfully commits.

• Atomic actions use a pessimistic concurrency control mechanism. Because most reconfigurations are expected to be relatively short-lived (on the order of a few seconds), the use of exclusive locks on model objects is unlikely to cause conflicts between concurrent reconfigurations to the same part of the application model, and any such conflicts will themselves be of short duration. Unfortunately this simple concurrency control scheme opens up the possibility of deadlock. At present I have not implemented any deadlock detection or avoidance mechanism, although it would be quite possible to do so, using any of a variety of well-understood algorithms such as [SN85, CKST89, CDK94c].

• Nested atomic actions are supported. These can be used to:
  
  – Improve throughput by allowing nested atomic actions to execute concurrently if possible.
  
  – Test multiple alternative reconfigurations within the scope of a single parent atomic action, aborting the nested atomic actions that prove unsatisfactory.
  
  – Support reconfigurations where some parts of the reconfiguration are allowed to fail.

• Nested atomic actions share the usual semantics of nested transactions:
  
  – Nested atomic actions cannot commit unless their enclosing parent atomic action also commits.
  
  – The parent atomic action can choose to commit even though some of its children have aborted.
  
  – Atomic actions at the same level of the nesting hierarchy can commit or abort independently of each other.

• There is the notion of a current atomic action for each thread of execution in the application model. There can be at most one top-level atomic action open in a given thread at any one time.

• An atomic action can be initiated at any time by any model layer object. If an atomic action is already in progress for the current thread, the object is simply added to the scope of the current atomic action. Otherwise, a new atomic action is initiated. Alternatively, an object may explicitly create a new nested atomic action within the scope of the current atomic action.

| Table 5.1: Properties and semantics of atomic actions. |
public class TransactionManager {
    // Class methods
    public static Transaction begin(ModelObject obj)
        throws TransactionException;
    public static Transaction beginNested(ModelObject obj)
        throws TransactionException;
    public static void commit(ModelObject obj)
        throws TransactionException;
    public static void abort(String why)
        throws TransactionException;
    public static Transaction current();
    public static boolean isCurrent(Transaction t);
}

Figure 5.5: Class djinn.transaction.TransactionManager.

no uncommitted atomic action in that thread. This implies that there can never be more than one
uncommitted atomic action in any given thread, and hence that application-level code should not
be permitted to directly initiate new (non-nested) atomic actions in violation of this rule. When an
atomic action is already running, any attempt by an object to initiate a new atomic action should
instead result in the object being added to the scope of the existing atomic action. Likewise, an
attempt to commit the atomic action should be ignored unless it comes from the component that
originally created the atomic action (any object in the scope of an atomic action can abort it, so abort
attempts should not be similarly ignored). Rather than relying on model objects to check whether
or not an atomic action is already running I have elected to encapsulate atomic actions within the
TransactionManager class, which performs all of the necessary checks and acts accordingly.
Figure 5.5 shows the public interface to this class.

An object wishing to initiate a new top-level atomic action should invoke the begin method,
passing itself as the obj parameter. If there is no atomic action running in the current thread, one
will be created and a reference to it returned; otherwise, the method simply returns a reference to the
current atomic action. In either case, the object obj is added to the scope of the atomic action. On
the other hand, the beginNested method will always create a new atomic action nested within
the current atomic action for the thread. A TransactionException will be thrown if there is
no current atomic action.\footnote{TransactionException is used to indicate failures in atomic action operations. It should be thrown by any
application-level method that uses atomic actions in any situation where the method’s actions cannot be completed
successfully. The usual result of throwing a TransactionException will be for the current atomic action to be
aborted, just as though the abort method had been invoked.}
public class Transaction {
    // Constructor
    Transaction(ModelObject obj);

    // Methods
    void add(ModelObject obj);
    void commit() throws TransactionException;
    void abort(String why) throws TransactionException;
}

Figure 5.6: Class djinn.transaction.Transaction.

The commit and abort methods have their usual transactional meanings. However, commit will not actually commit the changes made by the atomic action unless the obj parameter matches the object that first initiated the atomic action. In other words, an object should not necessarily expect to see the results of its reconfiguration activities as soon as it invokes commit—it has no way of knowing whether it is part of a larger transaction or not. The abort method will always immediately abort the current atomic action by throwing a TransactionException containing the why parameter, which should provide a human-readable explanation of why the atomic action was aborted. If the aborted atomic action is a nested child atomic action its parent need not necessarily abort as well; instead it should catch the TransactionException and decide whether the failure of one child atomic action warrants aborting the entire reconfiguration.

Class Transaction

Instances of this class encapsulate running atomic actions. A Transaction object has no public fields or methods and should be regarded by application-level code as an opaque reference to the current atomic action. The object does export a number of non-public methods that are invoked as necessary by the TransactionManager class (see Figure 5.6).

The constructor creates a new atomic action with the given model object obj in its scope, while the add method adds the given object to the scope of an existing atomic action. Adding an object to the scope of an atomic action involves:

1. Acquiring a lock on the object, to prevent other atomic actions from modifying it.
2. Making a copy of the object's internal state, in case the atomic action needs to be aborted and the object rolled back to its pre-reconfiguration state.

The commit and abort methods have the same behaviour as their counterparts in the TransactionManager class, except that commit will, in this class, always commit the atomic action.
class StateManager {
    // Class methods
    static void store(Transaction t, ModelObject obj);
    static ModelObject lookup(Transaction t, ModelObject obj);
    static void restore(Transaction t, ModelObject obj);
    static void restoreAll(Transaction t);
    static void delete(Transaction t, ModelObject obj);
    static void deleteAll(Transaction t);
}

Figure 5.7: Class djinn.transactionStateManager.

Class StateManager

This class has one very specific purpose: to make copies of the internal state of objects and to restore that state to the original objects if requested. Every object within the scope of an atomic action will have a copy of its state taken by this class in case the atomic action is aborted—in which case the object will be rolled back to the configuration it had before the atomic action was initiated, by replacing its current state with the restored copy. The methods exported by the class are shown in Figure 5.7. As with the Transaction class, these methods are not public and can only be accessed by objects or classes in the djinn.transaction package.

The store method takes a copy of the internal state of model object obj and stores it with any other state copies made by atomic action t, and is invoked when obj is added to the scope of t. The lookup method returns a reference to any copy of obj stored by t, and would be used by an atomic action that needed to refer to the original state of some object it had modified. If the atomic action t eventually commits successfully, its stored state copies are no longer required; in this situation the copies should be destroyed using the delete and deleteAll methods. The former removes the copied state of the given object obj while the latter deletes all of the copies stored by atomic action t. On the other hand, if the atomic action is aborted the objects within its scope must be rolled back to whatever states they were in when the atomic action was initiated. This is achieved by the restore and restoreAll methods. These methods invoke delete and deleteAll, respectively, after restoring the copied state.

Class LockManager

This class maintains lists of locks on model objects held by every running atomic action. Together with the Transaction class, this class is jointly responsible for implementing the concurrency control policy of atomic actions. Like StateManager, the LockManager class is not publicly accessible, but exports the methods shown in Figure 5.8 to classes in the djinn.transaction package.
class LockManager {
    // Class methods
    static void acquire(Transaction t, ModelObject obj, int type);
    static void release(Transaction t, ModelObject obj);
    static void releaseAll(Transaction t);
}

Figure 5.8: Class djinn.transaction.LockManager.

The acquire method locks the model object obj on behalf of atomic action t. The type argument specifies the type of lock to be acquired; the method consults a lock compatibility table [AA92, CDK94a] to determine whether or not the requested lock conflicts with any locks on the object already held by other atomic actions. Currently, only a single type of lock is supported: an exclusive lock that locks the object for both read and write operations. Only a single such lock may be held on a particular object at any given time.

If the lock cannot be acquired immediately, the acquire method will block until the requested lock can be taken successfully, usually once another atomic action has released its lock on the object. Note that this behaviour can lead to deadlock if there are any cycles in the locking dependency graph. The current implementation of atomic actions takes no steps to detect or recover from this situation.

The release method unlocks a single object previously locked by atomic action t, while releaseAll releases all of the locks acquired by t in a single operation.

5.3.2 Concurrency control

As I have mentioned previously, the short lifetimes and deterministic nature of program-driven atomic actions mean that a pessimistic scheme with exclusive read/write locks is an acceptable, if not optimal, approach to concurrency control for atomic actions. Additionally, it is a relatively simple scheme to implement, yet still provides all of the required functionality.

Locking adheres to the common strict two-phase locking protocol [CDK94a]: locks are acquired when objects enter the scope of an atomic action, but are not released until the atomic action is committed or aborted. More specifically, no lock can be released until all of the locks required for the atomic action have been acquired and, conversely, no more locks can be acquired once any lock has been released. It is the responsibility of the atomic action program to ensure that all necessary locks are acquired, by invoking the TransactionManager.begin method on every object that participates in the atomic action.
With regard to the transactional properties of Section 5.2.2, atomicity is guaranteed because the entire atomic action will be aborted if any part of it aborts. In the case of nested atomic actions, an aborted child will not necessarily abort the parent, but a child atomic action cannot commit unless its parent also commits successfully. The locking scheme ensures that the isolation property is maintained: in order to even read the state of a model object, an application must initiate an atomic action and acquire a lock on the object. This lock cannot be acquired while another atomic action holds a conflicting lock on the same object, thus it is impossible for one atomic action to read (or modify) the uncommitted state of another. The determination of consistency is application dependent; the mechanism for validating the consistency of a new configuration is described below.

5.3.3 Consistency validation

The final configuration or a reconfigured application is validated using a new abstract method added to the ModelObject class:

```java
public abstract void checkConfiguration()
    throws TransactionException;
```

This method is implemented by the Component, CompositeComponent and Port classes, and may be overridden by any other subclass of ModelObject that needs to perform class-specific consistency checks. The method should return normally if the target object's configuration is consistent, otherwise it should throw a TransactionException explaining why the configuration is inconsistent.

When an atomic action tries to commit, the Transaction class arranges for checkConfiguration to be invoked on every object in the scope of the atomic action. If any exceptions are thrown during this process, the atomic action is immediately aborted. Otherwise, the application is assumed to be in a consistent state and the atomic action is committed. The validation is performed in a "bottom-up" fashion; that is, proceeding from the atomic ports and components through progressively more complex composite components to the top-level objects of the atomic action. This allows higher-level objects to reject the decisions of those below them and invalidate a configuration even if lower-level objects believe it to be consistent.

5.3.4 Difficulties with Java implementation

My use of Java as the implementation language for the model object layer and atomic action framework is not without its drawbacks. Firstly, the StateManager class is almost impossible to implement as described above if one is restricted to using only the pure Java language. While it is trivial to make the initial copy of an object's state using the clone method provided by Java's basic
Object class and supported by all model objects, the same is not true of the operation to restore the state when an atomic action is aborted. The difficulty arises because the restore method needs to copy each field of the clone object into the corresponding field of the original, including the private and protected fields that are not generally accessible to the StateManager class. The Java Reflection API [Sun97a], a standard component of the Java class libraries since version 1.1, can enumerate all of the fields in a given object, but strictly enforces Java access control so that private and protected fields still cannot be read or written. Two possible solutions to this problem present themselves:

1. Every model object class must implement its own state-restoration method.

2. Provide a native (non-Java) code module to perform the state copying.

The first option is impractical as it would involve adding extra code to every model object class; this is clearly not consistent with my goal of making component implementations as simple and straightforward as possible. Neither is it a particularly elegant solution, exposing far too much of the internal logic of the atomic action mechanism. I would not claim that the second solution is terribly elegant either, but it does have the advantage of being generic and transparent to application programmers. I have thus chosen to implement the native code solution, as a small shared library compiled from C and dynamically linked with DJJNN at runtime. This library must of course be compiled separately for every hardware/operating system combination that DJJNN runs on. The state-copying code uses functions provided by the Java Native Interface (JNI) [Sun97b]; these offer a reflection-style interface to the fields and methods of Java classes from C or C++ code without the access-control restrictions of the pure Java reflection system. The JNI solution has the additional advantage that it should (in theory) compile and run without modification on any platform that supports a version 1.1 or later Java runtime.

More seriously, the standard Java runtime environment is not well suited to the provision of transactional operations. Transactional database systems generally do not require explicit action to add data items to the scope of a transaction. Rather, a program simply creates a new transaction, performs some operations, then attempts to commit the transaction. All accessed data will automatically be locked appropriately and added to the scope of the transaction. This is possible because the data is accessed through special read and write operations that carry out the necessary consistency control actions. Such an approach is impossible within DJJNN where programs can directly create and manipulate objects. Objects must explicitly add themselves to the scope of the atomic action; otherwise, there is no way for the system to know that the object is being read or written, nor that it must be locked and a copy of the state taken.

While it would certainly be possible to implement DJJNN in such a way that model objects were only able to be manipulated indirectly—say, through methods invoked on object handles—this would require a complete re-implementation of the framework and would, I believe, impose an unnecessary
burden on component and application programmers. My present solution is adequate for a prototype system to evaluate the atomic action concept, but would require significant improvement to be useful in a production system.

The PJama project [ADJ96, DAV97] has approached this problem from a different angle, by modifying the Java Virtual Machine (JVM) to directly support transactional operations. The system provides transactions supporting the full ACID properties and with a variety of concurrency control schemes, completely transparently to the programmer and without requiring modifications to any existing code. Such a solution would have been ideal for DJINN; however, at the time that I was implementing the atomic action mechanism for DJINN, PJama was only available for Java version 1.0 on a limited set of platforms. DJINN, unfortunately, requires a number of Java version 1.1 features—primarily RMI [Sun97c] and reflection [Sun97a]—so could not at that time be built on top of PJama.

5.4 Reconfiguring the sample applications

Section 4.4 has already shown how the sample applications from Chapter 3 might be implemented as DJINN components. This section extends the previous implementation of the application components to add facilities for reconfiguration. It is important to remember that any operation that acts to query or change the internal state of a component should be considered as a reconfiguration and performed within the scope of an appropriate atomic action; this is especially relevant when the state of active components is affected by the reconfiguration. Component creation must be treated as something of a special case—until the component is created there is effectively nothing to reconfigure and no object to place in the atomic action; nevertheless the act of creating a component consumes resources and must be validated in the same way as any other operation. To resolve this minor paradox, I simply require that any object wishing to create a component must do so within the scope of an atomic action created by itself or some higher level object. Then, if the atomic action is aborted, the original state will be restored and the creating object will no longer hold a reference to the new component, which can then be garbage-collected. In all other cases the component method itself must make the necessary calls to the atomic action subsystem.

For example, consider the powerOn method implemented by the ModelObject class and inherited by every object in an application model. This method is implemented as shown in Figure 5.9. Note first that that signature of the method is changed so that a TransactionException will be thrown if the atomic action is aborted for whatever reason. The first act of the method is to create

---

2 For instance, application programmers would not be able to use the new operator to directly instantiate a component. Rather, they would have to invoke a class method that created the object and returned an abstract handle to the caller.

3 An exception to this rule could perhaps be made for operation that cannot possibly fail or conflict with concurrent operations on the same or another component. However, since such operations are few and far between—certainly nothing that affects an active component meets these conditions—and the programming overhead of an atomic action relatively light, it seems reasonable to insist that all operations be encapsulated within atomic actions.
public abstract class ModelObject extends DjinnObject {
    // Other methods...

    // powerOn method with atomic action
    public void powerOn() throws TransactionException {
        Transaction t = TransactionManager.begin(this);
        if (modelState != OFF && modelState != STOPPED)
            TransactionManager.abort("Illegal state transition");
        addAction(new ActiveAction("powerOn", null));
        modelState = STOPPED;
        TransactionManager.commit(t);
    }
}

Figure 5.9: Implementation of the powerOn method.

A new atomic action; if the invocation of powerOn was within an existing atomic action then the
designated object (this) will simply be incorporated into the scope of the current atomic action
rather than creating a new one. The method then checks that the component is in a legal state to
perform the powerOn operation and aborts the atomic action if this is not the case. The following
two lines implement the actual reconfiguration:

- The addAction method queues an operation to be performed on the active counterpart
  of this object when the atomic action commits successfully—if the object does not have an
  active counterpart, the queued operation will be ignored.

- The state of the object is changed to STOPPED to indicate that it has been powered on.

Finally, the transaction manager is invoked again to commit the atomic action. At this point the new
configuration is checked for consistency and passed to the admission control mechanism. Assuming
that the configuration is consistent and that sufficient resources exist to realise it, any operations
queued by the addAction method will be executed on the corresponding active component(s).
On the other hand, if either of the consistency or admission control tests fail, the atomic action will
be aborted.

Most reconfigurations will follow a similar format to the powerOn method above, although generally operating at a higher level of abstraction. For instance, reconfiguration code in composite components usually does not have to worry about explicitly queueing actions on active components—this will be done as required by its atomic subcomponents. Figures 5.10 and 5.11 illustrate how a pair of typical reconfigurations on the surveillance application might be coded. Note that the code in these figures is incomplete, in the interests of clarity; in particular all of the error handling and
public class VideoServer extends CompositeComponent {
    // Existing methods...

    // Switch client to a new source
    public void switchSource(VideoClient client, Camera newSrc) throws TransactionException {
        Transaction t = TransactionManager.begin(this);
        Camera currentSrc = client.getSource();
        MulticastConnector oldConn =
            connectors.get(currentSrc);
        MulticastConnector newConn =
            connectors.get(newSrc);
        InputPort in = client.findPort("VideoIn");
        String clientHost = client.getHost();

        // Create new multicast endpoint
        OutputPort connOut = newConn.addSink(clientHost);
        OutputPort out = new ProxyPort(this, "clientHost", connOut);

        // Connect client to new endpoint
        in.disconnect();
        in.connectTo(out);

        // Shut down old endpoint
        oldConn.removeSink(clientHost);

        // Commit the atomic action
        TransactionManager.commit(t);
    }
}

Figure 5.10: Switching video sources in the surveillance application.
public class SecureSite extends CompositeComponent {
  // Existing methods...

  // Switch client from WaveLAN/Ethernet to GSM
  public void switchToGSM(VideoClient client)
  throws TransactionException {
    Transaction t = TransactionManager.begin(this);
    Camera source = client.getSource();
    MulticastConnector conn = connectors.get(source);
    InputPort in = client.findPort("VideoIn");
    String clientHost = client.host();

    // Set up the GSM link
    VideoClient gsmGate = GSMGateway.newClient(client);
    OutputPort out = gsmGate.findPort("VideoOut");

    // Make new connections
    in.disconnect();
    conn.removeSink(clientHost);
    switchSource(gsmGate, source);
    out.connectTo(in);

    // Commit the atomic action
    TransactionManager.commit(t);
  }
}

Figure 5.11: Switching networks in the surveillance application.
invocations of the basic state-changing methods have been omitted. Reconfigurations of the digital television editing application would be broadly similar in form.

5.5 Summary

Reconfiguration of DJINN applications is based around the two-level model-based approach to application structuring described in the previous chapter. Reconfiguration actions are performed firstly on the application model, and only applied to the active components of the application if the resulting configuration meets application- and system-specific structural and resource constraints. This chapter has presented a high-level view of the DJINN reconfiguration architecture, in particular those aspects relating to reconfiguration of application models.

Section 5.1 defined a reconfiguration as a transition of an application between two consistent states, where the notion of consistency is determined by an application-specific global invariant. Furthermore, it is a requirement of the multimedia application domain that, as far as possible, applications continue to run uninterrupted during reconfiguration and produce a minimum of unexpected output artifacts and glitches. The definition of reconfiguration is extended to include this idea of dynamic consistency which will be addressed in the following chapter.

Reconfigurations are implemented as atomic actions on the application model. Atomic actions are a transactional construct supporting the atomicity, consistency and isolation properties of traditional ACID transactions on databases. Two different styles of transactional reconfiguration—interactive and program-driven—were discussed, with quite different consistency control and conflict resolution requirements. Currently, only the program-driven style of reconfiguration has been implemented.

The later sections of the chapter outline the implementation of program-driven atomic actions in DJINN and give examples showing how typical reconfigurations on the two case study applications might be implemented. Chapter 6 will address the second layer of the reconfiguration system: applying configuration changes to the active components of an application while preserving the dynamic consistency of the system.
Chapter 6

Reconfiguration Scheduling

This chapter describes the second phase of the reconfiguration process outlined in Chapter 5—the updating of an application's active object layer to reflect the changes made to the application's model by a reconfiguration program. Chapter 5 has already shown how the model object layer of an application can be reconfigured atomically and consistently through the use of atomic actions. The most important consideration in the active layer update process is the maintenance of the application's dynamic consistency, a notion that I have so far defined only informally in terms of the temporal and data integrity of the application's media streams. My proposed solution to the problem of dynamic consistency is based upon the idea of scheduling the updates to the active object layer in such a way that the quality of the affected streams is maintained, while streams not participating in the reconfiguration are unaffected by it. The update schedule divides the update actions into two distinct phases: a non time-critical—but potentially resource-intensive—setup phase with minimal observable effect on the behaviour of the application; and a time-critical integration phase that completes the transition between the initial and final configurations of the active layer. A key factor in the scheduling algorithm is the tradeoff that must in general be made between the perceived "quality" of the reconfiguration, the time taken to complete the integration phase of the update and the quantity of additional resources—beyond those required for either the initial or final configuration—consumed during the update process.

The remainder of this chapter is structured as follows:

- Section 6.1 provides a detailed description of the active layer update process in terms of the components and paths involved, and motivates the need for scheduling of the update actions to ensure that dynamic consistency is maintained.

- Section 6.2 presents the development of an algorithm to derive and execute a schedule of update operations on the active object layer of an application, in such a way that the dynamic consistency of the application is maintained. The algorithm as discussed here is suitable for
Reconfigurations of a single path only; extensions to a subset of multiple-path reconfigurations are also discussed.

- Section 6.3 describes the implementation of the update scheduling algorithm within the current DJINN prototype.

A reconfiguration on the remote surveillance case-study application is used as a running example throughout this chapter. Experimental data from the execution of the algorithms described here on this sample application are presented in Chapter 7.

6.1 Motivation

Chapter 5 described the reconfiguration process up to the point of updating the active object layer of the reconfigured application, the final step that will make the effects of the reconfiguration visible to observers of the application's media streams. To recapitulate, by the time the reconfiguration reaches the point at which the active layer updates must occur:

- The atomic action controlling the reconfiguration is in its commit phase; that is, the commit method has been invoked on the atomic action and it must either: atomically commit the changes made by the reconfiguration to both the model and active layers of the application, or abort the atomic action and return the participating objects to the state they had before the reconfiguration was initiated.

- The new configuration of the application model has been subjected to, and passed, an admission control test. A sufficient quantity of resources have been reserved to meet the QoS requirements of the new configuration.

- The new configuration is known to be statically consistent, under the application's definition of consistency.

To complete the reconfiguration, the active object layer must be updated to reflect the final configuration of the reconfigured model layer. In terms of the hierarchically structured model layer, this means that the configuration of each component and port in the active layer must be changed to match the configuration of the corresponding component or port in the model layer. For components, the configuration consists of:

1. The set of ports attached to the component

2. The activity state of the component (STOPPED, RUNNING, etc.)
3. QoS settings

4. Other component-specific settings

while for ports the configuration includes:

1. The component that the port is attached to
2. The other port that the port is connected to
3. The activity state of the port
4. In the case of input ports, the media data and event handlers for the port

Clearly, all of these settings can be determined by querying the state of the relevant model objects. Furthermore, since the initial pre-reconfiguration state of the model objects is stored by the atomic action StateManager, it is trivial to determine how the configuration of any given model object has changed due to the reconfiguration, and thus what changes should be made to the corresponding active object.

However, a further requirement of the reconfiguration is that the active layer update be dynamically consistent. From Section 5.1.4 the necessary conditions for dynamic consistency are:

1. The reconfiguration must be performed atomically. For the model layer, the atomicity property ensures that the application is not left in an inconsistent state partway between the initial and final configurations. The same consideration applies to the active layer, with the additional complication that it is generally not practical to reverse changes to active objects—once the object has processed some media data and sent it on downstream, the data cannot be "taken back". Likewise, because the internal state of an active object may be dependent on the data it has already processed, keeping a backup copy of the object's state—as is done by the model layer—would essentially mean running two copies of the object in parallel for as long as necessary to complete the reconfiguration. Resource considerations will generally make such an approach impractical. Thus, no changes should be made to the active layer unless it is certain that all of the changes relating to a particular reconfiguration will succeed. This condition is met as long as the final configuration of the model is consistent, and provided there are no crashes or hardware failures during the update.

2. No corrupted or otherwise erroneous data should be produced by the actions of the reconfiguration. For example, the internal buffers of an MPEG decoder must be flushed before it begins decoding a new stream, otherwise corrupted frames will be generated. I will refer to this specific requirement as data consistency.
3. Likewise, the temporal properties of streams affected by the reconfiguration should not be violated. For example, there should not be a long pause or dramatic change in frame rate when switching a video player from one input stream to another, unless these are called for by the reconfiguration. This requirement will be referred to as temporal consistency.

4. Objects not directly participating in the reconfiguration are not affected. That is, unless the configuration of a given active object is actually being changed, that object should not be aware that a reconfiguration has taken place.

Difficulties arise in the reconfiguration of the active layer by its very nature: the objects undergoing reconfiguration are actively engaged in processing streams of continuous media and should be able to continue this activity even while the reconfiguration is in progress. Contrast this situation with the execution of an atomic action on the model layer of an application. Once the new configuration of the model has passed the necessary consistency and admission control tests, the new configurations of individual model objects may be committed in any order and over an arbitrarily long period of time. The only potential adverse effect is that other reconfiguration programs may have to wait for access to the reconfigured objects. These properties hold because none of the changes to the model are made visible—by releasing the locks held by the atomic action—until all of the changes have been applied. Thus, external observers will see all reconfiguration actions occurring simultaneously, regardless of the order in which they were actually carried out.

Active objects, on the other hand, should be able to continue processing during a reconfiguration; they cannot in general have their operation arbitrarily suspended without violating the dynamic consistency of the application. Furthermore, the order and timing of updates to active objects is important. Consider the simple reconfiguration of a video player application shown in Figure 6.1. For instance, if the file reader component starts producing M-JPEG data before the new decoder has been connected and started, the existing MPEG decoder will probably be forced to discard the unexpected input, such that the initial M-JPEG frames will never be displayed. Likewise, the file reader must have ceased to generate MPEG data before the M-JPEG decoder is started. Neither of these outcomes will be prevented by the corresponding atomic action on the model layer, as each still results in a statically consistent configuration, despite the obvious loss of dynamic consistency.

The previous example is trivial enough for an application programmer to see the problem and develop an appropriate solution within the model layer reconfiguration code. For example, if the file reader is shut down when the reconfiguration begins and not restarted until the new decoder is in place, there is no chance for any incorrect data to reach the decoder. However, this approach could be considered somewhat fragile in its reliance on application programmers to both spot potential dynamic inconsistency—recall that application programmers are not supposed to concern themselves with the active layer at all—and implement a correct solution. I would conjecture that this is an unlikely scenario for all but the simplest reconfigurations. Perhaps more seriously, it is not necessarily possible to resolve all dynamic consistency issues from the model layer. If the file reader in
the example is stopped then, depending on how quickly the rest of the reconfiguration proceeds, it may be some time before it is restarted, resulting in a pause in video playback at the display. In this particular example, the reconfiguration can be further reordered to minimise the unwanted pause; nevertheless, this will not be feasible in all situations and in any case is not something application writers should be concerned with.

Therefore, I believe there is a need for a generic mechanism to manage the updating of active objects in the commit phase of an atomic action. Such a mechanism should control the order and timing of updates to individual objects so as to preserve the dynamic consistency of the running application, while ensuring that the active layer reaches the final, stable configuration dictated by the atomic action. Putting this another way, the mechanism should compute and execute a schedule for the update actions applied to the active layer. In earlier work I have referred to this mechanism as "smoothness", after the so-called "smoothness property" that refers to the observed behaviour of an application whose dynamic consistency is maintained during reconfiguration: intuitively, the reconfiguration proceeds smoothly if there are no observed glitches, lost or corrupted data, and the application successfully reaches its new statically consistent state. The scope of the smoothness concept has broadened somewhat to incorporate resource usage and larger issues of reconfiguration timing, but the general principle remains the same.

6.1.1 High-level view of smoothness

The definition of smoothness given above is somewhat subjective, relying to a certain extent on individual users' perception as to what constitutes an "observed glitch". It would be useful to have a more precise, quantitative measure of the dynamic consistency achieved during a reconfiguration, as a basis for comparing different scheduling strategies. Coupled with user and application constraints on reconfiguration "quality", such a measure would also allow the scheduling mechanism
to determine how much effort to expend on optimising a particular reconfiguration—a sub-optimal solution might provide sufficient quality for the target application at much less computational expense.

One might argue that dynamic consistency is only maintained across a reconfiguration if no data is lost or corrupted and if each object involved in the reconfiguration continues to operate within its QoS guarantees and resource constraints at all times. Leaving aside the fact that QoS and resource settings might change during the transition from initial to final configuration, I would contend that this argument is far too inflexible for many situations. Given that an application is undergoing reconfiguration, users may be prepared to tolerate a certain amount of degradation in QoS, or temporary violation of resource constraints, if it can be guaranteed that the reconfiguration will succeed. Alternatively, users may not even notice a loss of quality around the reconfiguration: there is a well known technique called temporal masking that takes advantage of the fact that the human visual system cannot discriminate even a large drop in the quality of a video stream either side of a scene change (The BBC's ATLANTIC switch uses this technique). For these reasons I believe it is feasible to allow a controlled reduction in stream quality during reconfiguration, provided that this is compensated for by the application remaining within its resource constraints, completing the reconfiguration in a shorter time, or indeed being able to complete it at all. Similarly, short-term resource overruns may be acceptable in return for a smoother or faster reconfiguration. Resource usage can be modelled and measured; likewise changes in the QoS characteristics of an object or stream.

While it may be necessary to track the QoS and resource behaviour of each active object individually to ensure the success of a reconfiguration, users are unlikely to want to specify desired reconfiguration quality at such a fine-grained level. Rather, they are concerned with maintaining a certain level of end-to-end performance during the transition. The D3NN framework already has a construct that expresses static end-to-end properties: the path (see Section 4.2.5). Thus, I have chosen to use paths as the primary application-level unit of reconfiguration. An application specifies its desired reconfiguration quality as path attributes indicating, for example, the allowed change in end-to-end latency or CPU usage during the reconfiguration. The scheduling algorithm is then able to analyse changes in the structure of the path and the configuration of its individual components to compute a schedule that meets the specified constraints. Of course, not every part of an application is necessarily part of a path: what happens to these objects during reconfiguration? Clearly they must still be reconfigured, but in the current algorithm they are ignored for scheduling purposes. The fact that paths are intended to encapsulate the "important" aspects of an application's processing provides some justification for this decision; however it remains to be seen whether this approach will be sufficient in the general case.

The use of paths as the basis for reconfiguration is also advantageous in reducing the space of different reconfigurations that must be handled by any scheduling algorithm. Figure 6.2 shows the three basic structural reconfigurations that are possible on a single path: insertion, deletion
and replacement, where A, B, C, etc. represent contiguous sub-regions of the path. Any more complex reconfigurations can be composed from this set of building blocks, with the total number of possible reconfigurations scaling exponentially with the length of the path—fortunately, none of the algorithms described later have a complexity beyond \(O(N^2)\). Note that a structural change to a path object does not necessarily correspond to a physical connectivity change in the active layer; the application may simply be changing the routing of a path through already connected objects. Similarly, a reconfiguration may not involve any structural change to the path itself if the application is, for example, just changing the settings of existing path elements. However, reconfigurations that include structural change to the underlying active layer are the most interesting from a scheduling and dynamic consistency point of view.

6.1.2 Dynamic consistency as a continuous variable

Ultimately, DJINN’s target applications operate in a resource-constrained world: there are overall hard limits on the amount of processing or communication an application can perform in a given time interval, and all of the resources consumed in doing so do have a cost, even if only in the opportunities denied to other applications. These limits become particularly relevant when considering reconfiguration, where control of resources must be transferred between the initial and final configurations of the system. Very often, the final configuration will have a different resource mix, although it may well be using many of the same physical resources, allocated to different objects. Consider a reconfiguration that causes the replacement of a sequence of components between two fixed points, such as a switch from MPEG to H.263 coding in the surveillance application. When resources are scarce it will be necessary to completely shut down the old configuration before the new components can operate at their guaranteed service levels. On the other hand, if there is a
surfeit of resources both configurations can be run in parallel for as long as necessary to ensure a smooth transition.

This example illustrates the point that dynamic consistency and smoothness are not simple binary properties; rather they occupy a continuum where reconfiguration quality can be traded off against resource usage according to user and application requirements, available resources, and the characteristics of the active objects involved. Increased smoothness (that is, fewer violations of dynamic consistency, and an intuitively higher quality reconfiguration) will generally require the use of more resources, and vice-versa, although the space of potential operating points will obviously vary per reconfiguration. The two extremes are those mentioned in the example above: with infinite resources, perfect smoothness can be achieved by running both configurations in parallel, whereas if there are no free resources the initial configuration may have to be shut down before the final configuration can be started, with a consequent loss of dynamic consistency. Between these boundary cases lie a range of compromise solutions.

Updates to the active layer do not happen instantaneously, so it is useful to consider the time taken to complete a reconfiguration as a third "axis" to the reconfiguration tradeoff. The latency of operations such as creating or powering on a component are such that there will always be some delay between a user saying "reconfigure now" and seeing the results of her reconfiguration. Resource constraints can increase this delay; suitable scheduling may be able to reduce it, as the following section will show. In any case, each potential smoothness/resource operating point for a given reconfiguration will have an associated latency, which must also be taken into account in choosing an appropriate schedule for the transition.

6.2 Algorithms

This section presents the algorithms I have developed to provide smooth, dynamically consistent reconfigurations of a single path in a DJINN application. There are two algorithms, which are run during the commit phase of an atomic action to reorder and schedule the updates to the active objects. The first divides the set of updates into several stages, allowing parts of the reconfiguration to be carried out in advance to reduce the overall latency of the reconfiguration and improve smoothness. The second computes and executes a more fine-grained schedule for the updates in the final stage of the reconfiguration, to keep the observed transition between initial and final configurations within the resource and smoothness constraints specified by the path.

Throughout this section the operation of the algorithms will be illustrated through the example reconfiguration shown in Figure 6.3. This is a simplified version of the reconfiguration on the remote surveillance application from Section 5.1 (Figures 5.1 and 5.2), where an adaptive video connector switches from sending MPEG video over a high-bandwidth WaveLAN to sending lower quality H.263 frames across a GSM link. Note that any objects shared between the two path configurations
are only shown once in the diagram, and objects or connections that do not exist before the reconfiguration begins are drawn with dashed lines. Each object is annotated with the reconfiguration actions to be performed on it.

6.2.1 Two-stage updates

We can make a number of useful observations regarding the progress of a reconfiguration on the active layer of a DJINN application. Firstly, it is generally true that the major changes in an application’s behaviour occur at component state transitions, such as the STOPPED→RUNNING transition initiated by the start method. Other reconfiguration actions on a given component are typically in preparation for an impending state change, or minor tweaks to adjust the behaviour of a running component. In addition, state changes are often associated with structural and connectivity changes.

Secondly, recall that a component does not occupy or utilise any of the resources reserved for it until it undergoes an OFF→STOPPED transition (power on) and that even at this stage resources such as CPU bandwidth may be assigned to the component’s exclusive use but not actually consumed until the STOPPED→RUNNING transition begins media processing activity. In addition, many components can be designed so as to have no user-observable effects until they move into the RUNNING state, provided that sufficient resources are available. Thus I contend that it is feasible, for many reconfigurations, to carry out some proportion of the active layer updates in advance, without any observed effects and before making any changes to the existing, running configuration.

Thirdly, reconfigurations are almost always triggered in response to some external event, such as: the passage of a particular instant in real (or virtual) time; the appearance of a particular media element in a stream; user interaction; or a request from the middleware layer. In many cases, the application will have some inbuilt knowledge or receive some advance warning that a reconfiguration is going to occur in the near future. It may then be able to take advantage of the previous point to begin the reconfiguration before it is requested, minimising the amount of work to be done when the trigger event is actually received and decreasing the apparent execution time of the reconfiguration. Clearly, in many cases—software or hardware failures, for example—there may not be any advance warning of the need to reconfigure, but where such information is available it seems reasonable to make use of it.

Finally, if we step back and view the reconfiguration from the level of the path(s) involved, where there are state changes in the path members, these will generally all be in the same "direction"; that is, the entire path path moves towards a more or less "active" state, where the level of activity is as defined by the ordering of active object states in Table 6.1. The situation can be complicated

\footnote{This point applies primarily to pure software components, or those utilising shared hardware resources. A component that requires exclusive access to some device cannot be powered on until any existing use of that device has relinquished control; this qualifies as a user-observable effect since any output produced by the previous user of the device must then cease.}
<table>
<thead>
<tr>
<th>Activity level</th>
<th>State(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (least active)</td>
<td>DELETED, not activated yet</td>
</tr>
<tr>
<td>1</td>
<td>OFF</td>
</tr>
<tr>
<td>2</td>
<td>ON</td>
</tr>
<tr>
<td>3 (most active)</td>
<td>RUNNING, PAUSED</td>
</tr>
</tbody>
</table>

Table 6.1: Path activity states.

somewhat when the reconfiguration involves structural reconfiguration as well as state changes: if objects in the initial configuration are being shut down while objects in the final configuration are started up it might appear that the path is moving in both directions at once. However, the overall effect of the changes is that the path itself remains in the same activity state, while certain well defined segments of it have moved into more or less active states.

These observations have led to the development of two-stage updates. The goal of this mechanism is to separate out the active layer update actions that can be performed:

1. without affecting the ongoing operation of already running paths
2. within the available resources

into a setup stage and execute these in advance, before the reconfiguration trigger event, wherever possible. The remaining actions are for the integration stage which will be executed once the trigger event is received. There is nothing magic about the choice of two stages; it is certainly possible to devise a more fine-grained partitioning of active layer updates with—in the boundary case—as many states as there are update actions. However, as the number of states increases their scheduling becomes more complex, negating the benefits of path-level reconfiguration. The existence of reconfiguration trigger events provides an obvious natural boundary for a two-stage scheme and, more pragmatically, this approach is far easier to implement and analyse.

Figure 6.4 illustrates the effect of the setup and integration stages on the observed quality of a reconfigured path, where "quality" is an application-defined metric typically related to the QoS attributes of the path. The setup stage ideally has no observable effect on the operation of the path components, so quality will remain unchanged. During the integration stage, resource shortages may occur while components from both configurations are running simultaneously, or there may simply be unavoidable delays in starting up certain components or getting data out of them. Thus, there is likely to be a reduction in perceived quality during this stage. It may be possible to reduce the size of the quality loss by rescheduling the integration action, or by adjusting the boundary between the two stages; these optimisations are discussed below in Section 6.2.2 (page 164).
Relationship to atomic actions

Atomic actions continue to operate essentially as described in Chapter 5. The two-stage update algorithm does not come into play until the very end of the atomic action's commit phase. By this point the reconfiguration has passed admission control and static consistency tests, so it is known that all of the update actions will succeed, although some resources occupied by the existing configuration may need to be released first. The actions in the setup stage can be executed immediately; the atomic action is then suspended until a suitable, reconfiguration trigger event is received. Then, the integration stage actions will be executed according to their precomputed schedule.

The main advantage of the multi-stage approach is in allowing time-consuming parts of an atomic action to be committed while the initial configuration of the application continues to run, thus masking at least some of the latency of the reconfiguration. This is similar to a restricted version of Kaiser and Pu's split- and join-transaction [KP92] which allow an ongoing transaction to be split into two or more mutually serializable sub-transactions that can be committed or aborted independently of each other. One use of a split-transaction is to commit some work early; unlike a DJINN atomic action there is no requirement that the rest of the transaction ever be committed.

One drawback of an early partial commit of an atomic action is that any additional resources claimed by the committed actions are not in fact reserved for their use until the reconfiguration has executed completely. Only resources that were otherwise free may be occupied in this way; nevertheless, other reconfigurations may be prevented from committing because insufficient resources are available during their admission tests. Likewise, the model objects involved in the partly-committed

---

2The trigger event may, of course, have already happened, if there was insufficient advance warning of the reconfiguration to complete the setup stage before the event arrived. In such cases, the atomic action will not be suspended.
reconfiguration will remain locked until the integration stage has completed; this too may prevent other atomic actions from running. Unfortunately there is at present no practical way of aborting an atomic action once it has made changes to the active layer, so there is no realistic option other than to wait for the reconfiguration to complete. The problem can be partially alleviated by scheduling the execution of the setup stage so as to minimise the time spent waiting for the trigger event, or by reducing the quantity of resources claimed by the setup stage actions. The former may prove difficult if the timing of the trigger is not precisely known, while the latter can impact negatively on the smoothness and integration latency of the reconfiguration.

Determining stage boundaries

The most crucial aspect of the two-stage update algorithm is defining the boundary between the setup and integration stages for each object along the reconfigured path. This decision directly affects the latency and resource needs of the setup stage and influences the schedule computed for the integration stage.

It is worth mentioning that the actions of a reconfiguration program on the model impose a partial ordering on the corresponding active layer actions. Within an individual active object, the actions should in general be executed in the same order they were performed on the model object. Given suitable detailed knowledge of the application and/or object in question, it might be possible to reorder these actions in such a way that the final configuration remains unchanged. However, since it is not in general possible to change the ordering of an action relative to a state transition, nor to re-order state transitions with respect to each other, any such efforts are of dubious value. All of the actions enqueued by the reconfiguration program are timestamped so it is trivial to obtain a global ordering across the entire atomic action. Apart from connectivity changes and the creation or deletion of ports, actions on different objects are independent of one another; thus these actions can be freely re-ordered. However, in practice smoothness considerations that require, for example, that a component is started before its downstream neighbour, limit the scope for arbitrary re-ordering.

The inputs to the two-stage update algorithm are the initial configuration $P$ and final configuration $P'$ of a path:

\[
P = \{p_1, p_2, \ldots, p_N\}
\]

\[
P' = \{p'_1, p'_2, \ldots, p'_M\}
\]

where some of the elements of $P$ may map $1 \rightarrow 1$ onto elements of $P'$; this is typical but not necessary. Any element of $P$ or $P'$ may require reconfiguration of the corresponding active object.

The first step in determining the inter-stage boundary is to compute the shared and divergent re-
for each element of initial path configuration
   search for a matching element in final path configuration
   if a match is found
      maybe create a new divergent final path region
      move all elements of final path configuration prior to the match
      onto the final path region
      maybe create a new shared initial path region
      maybe create a new shared final path region
      move matching element to initial path region
      move matching element to final path region
   else
      maybe create new divergent initial path region
      move non-matching element to initial path region
      end if
end for
for each remaining element of final path configuration
   maybe create new divergent final path region
   move element to final path region
end for

initial/final path region The last element of the path region list for the initial/final path configuration. Each member of the list may be a shared or divergent region.
maybe create If the last element of the named region list is not of the specified type (shared or divergent), create a new region of that type and append it to the list. The new region becomes the current region for that list.

Figure 6.5: Computing shared and divergent path regions.

regions with respect to the two path configurations, as illustrated by the pseudo-code in Figure 6.5. For the example reconfiguration, the shared and divergent regions are as shown in Figure 6.6. Here we have regions of overlap at either end of the paths, with a divergent region in between terminated by the ports of the MPEG and H.263 codecs. Several reconfiguration properties can be inferred from the pattern of overlapping regions, irrespective of the underlying active object structure:

- If the endpoint of an overlapping region and the adjacent endpoint of a divergent region are both ports, then there must have been a structural reconfiguration at this boundary, since the port in the overlapping region is connected to a different port in the final reconfiguration.

- If, on the other hand, the endpoint of an overlapping region is a component while the adjacent divergent region endpoint is a port, then this boundary simply represents a re-routing of the path through a different port attached to the same component. There is no structural reconfiguration here that affects both the initial and final configurations.

\footnote{A divergent region is defined as a contiguous sub-region of either path configuration that does not occur in the other configuration. Shared regions, on the other hand, are sub-regions that exist in both configurations. Note that two shared regions occurring in reversed order in the other configuration will not be detected as such; one will be marked as shared and the other as two divergent regions at opposite ends of their respective configurations.}
There are no other structural reconfigurations within the overlapping region.

There may be structural reconfigurations within divergent regions; however, these will affect either the initial or final configuration of the path, but not both.

The next step in the process is to identify the regions of each path that actually require reconfiguration. This is a trivial task, since any model object on the path with pending active layer actions is, by definition, awaiting reconfiguration. Figure 6.7 shows the extent of the reconfigured regions for the example reconfiguration, while Figure 6.8 presents pseudo-code for the computation. Note that these regions can be computed separately for each path configuration. There is no necessary relationship between the reconfigured regions and those shown in Figure 6.6 although I anticipate that, as in this example, there will generally be some correspondence. Here, the reconfigured region covers the divergent parts of both path configurations and extends as far as the endpoints of the two overlapping regions.
for each element of path configuration
    if element requires reconfiguration
        maybe create new reconfigured path region
    else
        maybe create new static path region
    end if
move element to path region
end for

Figure 6.8: Computing reconfigured path regions.

What can we infer from these regions, and their relationship to the overlap/divergence regions discussed above?

- The regions requiring reconfiguration do precisely delineate the extent of changes to the active layer, but do not provide any additional clues as to whether the changes are structural or otherwise. Inspection of the actions themselves is required to find this out.

- The initial and final states of the reconfigured objects can be examined to determine the “direction” of the reconfiguration, in terms of activity states. As mentioned earlier, I assume that objects across all of the overlapping path regions will move in the same direction, as will the reconfigured objects in each divergent region. It is easy to see that these assumptions hold for the example reconfiguration: the objects in the two shared regions remain in the same state (RUNNING); those in the divergent initial path region move from the RUNNING state towards DELETED; while the object in the divergent final path region will be created and move towards the more active RUNNING state.

- As it turns out, the intersection between reconfigured and overlapping regions is not particularly useful. However, where a reconfigured region crosses the boundary between overlapping and divergent path regions this does at least suggest that:
  - There is high probability of structural reconfiguration in this region.
  - This region will be important in maintaining the dynamic consistency of the path.

The algorithm is now at the point where the setup/integration stage boundary for each object can be computed. The various regions are not used directly in this process; however, they are important to the integration scheduling algorithm and are used by several optimisations to the two-stage update algorithm described below.

Because the setup stage is not allowed to cause any observable effects, there are number of specific actions it cannot include:
for each region in path configuration
  if region requires reconfiguration
    for each object in region
      if object state ≥ RUNNING
        stage ← INTEGRATION
      else
        stage ← SETUP
      end if
    end for
  end if
end for

Figure 6.9: Computing setup/integration stage boundary.

- Any action on a currently running object must be excluded, since it may have an observable effect, but there is currently no way for the algorithm to know whether or not it actually will.

- Likewise, no object should undergo a ON→RUNNING or RUNNING→ON state transition as part of the setup stage.

- An object should not make the OFF→ON state transition unless there are sufficient free, unallocated resources available to support it.

All other actions are legal candidates for inclusion in the setup stage.

For each reconfigured object on the initial and final configuration of the path P, the setup/integration stage boundary is calculated as shown in Figure 6.9. The input to this code is a list of path regions obtained by combining the static/reconfigured regions from Figure 6.7 with the shared/divergent regions from Figure 6.6. For example, an object that is part of a shared region and also happens to require reconfiguration will become part of a new shared-reconfigured region. The computation is run separately on the regions of the initial and final path configurations, with shared regions are excluded from the run for the final path configuration to avoid calculating their stage boundaries twice. Figure 6.10 shows the stage boundaries for the example reconfiguration, indicated by the heavy lines separating actions on the diagram.

The setup stage may be executed immediately, or held until a time closer to the expected arrival of the integration trigger event. In any case, there will be no visible effect when it is executed, although resources may be consumed as discussed previously. Note that the setup actions for each component will be executed in the same order they were performed on the model by the original
Figure 6.10: Setup/integration actions for basic scheduling algorithm.
atomic action. While it might be desirable to reorder or overlap the actions so that the setup stage executes faster and/or with lower resource cost, there is no general way of determining whether any arbitrary reordering will lead to the same consistent final state.

6.2.2 Integration scheduling

The integration scheduling algorithm comes into play once the active layer update actions for a reconfiguration have been divided into setup and integration stages for each affected object. The aim of this algorithm is to compute a schedule for applying the updates in the integration stage that:

1. Maximises the dynamic consistency of the integration stage, i.e. maximise smoothness.
2. Minimise the usage of resources during the integration stage, beyond those already allocated to the reconfigured objects by the admission control system.
3. Minimise the time taken to complete the transition to the final configuration.

Clearly there can be conflict between these goals, and it will not in general be possible to meet them all simultaneously. The user or application must specify—as attributes to the paths being reconfigured—how tightly constrained smoothness, resource usage and time should be for each reconfiguration. Increased flexibility in these parameters will give the scheduling algorithm more room to manoeuvre; that is, more options to choose from in determining a schedule that meets the requirements of the application within the constraints imposed by the environment.

I will present the scheduling algorithm in two parts: firstly, a simple algorithm that is guaranteed to result in the correct final configuration, but may trigger resource overruns and loss of dynamic consistency; and secondly, optimisations to this basic algorithm that illustrate how quality, resource usage and time can be traded off against one another to produce a subjectively better reconfiguration.

Simple scheduling

Rather than scheduling the integration stage of each active object individually, I treat the initial and final configurations of the path as sequences of regions to be reconfigured. The regions are derived from the regions of reconfiguration already computed by the two-stage update algorithm (Figure 6.7), further divided where they intersect with the boundaries of overlapping path regions (Figure 6.6)—that is, the regions used in the stage boundary calculation of Figure 6.9. For example, Figure 6.11 shows the “integration regions” for the example reconfiguration. In this case there are four regions, two of which are shared between the initial and final configurations.

Integration begins when the designated trigger event is received and proceeds along each path configuration from the most upstream to the most downstream region. Where the configurations overlap, any reconfiguration regions within the overlap cannot be reconfigured until all of the upstream
regions on both configurations have completed their integration. Where the path configurations are divergent, regions can be integrated in parallel. Figure 6.12 gives some pseudo-code for this process. In the example reconfiguration, region A is the first region encountered on either path configuration. Once the objects in region A have been reconfigured, integration of regions B and C can proceed in parallel. The integration of region D must wait until both regions B and C have completed before it can begin.

Within each reconfigured region, the integration stage actions of each object are executed beginning with the most upstream object and moving downstream to the end of the region. The actions are executed in the same order that they were on the corresponding model object and the integration of each object must be completed before the next is started. This could be viewed as a somewhat conservative strategy, since in many cases the integration of different objects can be overlapped without affecting the end result of the reconfiguration—the extra parallelism gained from this approach could significantly speed up the integration process. However, it is difficult to determine in a generic way which actions can be safely run in parallel, so I have not pursued this option as yet.

This basic algorithm will certainly ensure that all of the active layer updates are executed. However, it has a number of flaws that render it unsuitable for many reconfigurations:

- An object will not be powered on during the setup stage if this would consume more resources than are currently available. Unfortunately, such objects will then be powered on during the integration stage regardless of the resource situation—although the necessary resources will have been reserved by the admission control mechanism, they may be occupied by objects in the initial configuration and thus not available until these objects are powered off. There are at least three possible courses of action in this situation, although none are completely satisfactory:
in parallel
  for each member of initial path region list
    if region requires reconfiguration
      wait for integration of upstream regions to complete
      for each object in region
        for each reconfiguration action on object
          execute action
        end for
      end for
    end if
  end for
for each member of final path region list
  if region requires reconfiguration
    wait for integration of upstream regions to complete
    for each object in region
      for each reconfiguration action on object
        execute action
      end for
    end for
  end if
end for

Figure 6.12: Basic integration scheduling process.

1. Suspend the integration of this region until the required resources become available. This is a simple solution, but will increase the time to complete the reconfiguration and potentially cause violations of dynamic consistency.

2. Acquire as many resources as possible now, continue with the integration and obtain the missing resources later. This will allow the integration to proceed, but the object may fail or behave incorrectly if it does not have its full quota of resources.

3. “Steal” resources from an object that will be giving them up anyway, as part of its integration stage. This has the same problems as the previous solution, and would be difficult to implement.

- The perceived smoothness of the integration is largely dependent on the time taken to integrate each region. In the example reconfiguration, ideally region C should complete its integration immediately after region B, so that the first H.263 frame reaches the display immediately after the last MPEG frame. Whether this happens or not depends on the time taken to start up the H.263 codec and GSM link versus shutting down the MPEG codec and WaveLAN link. In practice, even if all of the components were powered on during the setup stage, it is likely to take longer to start up the final configuration than to shut down the initial configuration, so there will be a period when no data is received by the display at all—this may or may not constitute a smoothness violation, depending on the path attributes in force.
for this reconfiguration.

- Any `connect` operations performed in the integration stage will fail if the port being connected to does not yet exist. This should happen very infrequently—ports will usually be created during the setup stage as necessary—but the situation can occur if, for example, the port must be created on a downstream component that is already running. The problem can in fact be solved quite simply by ensuring that connections are always made from the downstream port to the upstream one, rather than the other way around. If all `connect` operations are checked by the update algorithm and reversed if necessary, programmers can continue to write their reconfiguration programs in whatever way is most convenient for them.

Optimisations to the basic algorithm

The basic algorithm described above is adequate, in the sense that reconfigurations using it will run to completion, but may still exhibit a loss of temporal integrity in several common situations. However, if the scheduler has access to other more detailed statistics on the behaviour of each active object, it is possible to optimise the scheduling such that most of these violations can be eliminated. A lot of useful data is available as a side-effect of generating the QoS model for an active object class on a particular platform, including:

- The time taken to complete each update action on the object.
- The set of resources consumed or released by each of those actions.
- The latency between input and output ports of a component; that is, the time that elapses between a given media element entering a component through some input port and a media element derived from the first emerging from an output port on the same component.
- As an extension of the previous point, the latency across the fragmented objects making up a connector component.
- The jitter—the maximum variation in latency—between input and output ports of a component.

Given this information, at least two different classes of optimisation may be attempted:

1. Adjustment of the setup/integration stage boundary for individual objects.
2. A more fine-grained scheduling of the integration stage of the reconfiguration.
Changing the setup/integration boundary allows a reconfiguration to trade off resource usage during the setup stage against the smoothness and timeliness of the integration stage. Consider again the reconfiguration shown in Figure 6.3. The predictive coding used by the MPEG and H.263 formats means that the respective codec components will have relatively high latency; that is, several frame times typically elapse between putting a raw frame into the encoder and the coded version emerging, and vice-versa at the decoder. Under the scheduling algorithm described above, this delay will be experienced during the integration stage, when the first frame data reaches the codec components of the final configuration, thus increasing the time to complete the reconfiguration \( t_{\text{integrate}} \) in Figure 6.4.

If available resources permit, some of this codec delay can be hidden by starting the final configuration running before the integration trigger event is received. In terms of the scheduling algorithm, this means moving the setup/integration boundary such that the setup stage now includes the STOPPED→RUNNING transition of the H.263 codec components. Obviously, the boundary must also be adjusted for other components in the final reconfiguration, to ensure that the data makes it as far as the H.263 decoder. This change does not, of course, eliminate the inherent latency of the H.263 encoding and decoding algorithms; frame will still emerge from the encoder \( N \) frame times after they entered it. However, what it does do is ensure that the decoder is already producing frames when the integration stage is triggered, thus reducing \( t_{\text{integrate}} \) to the time taken to execute any remaining integration stage actions. Figure 6.13 illustrates how this optimisation would affect the stage boundaries computed for the example reconfiguration.

A reconfiguration using this technique must still respect the “no observable change” property of the setup stage and ensure that any additional setup actions do not cause resource shortages. Thus in this case the final path configuration can be started running as far as the H.263 decoder—resources permitting—but the final connection to the display cannot be made until the integration stage. In addition, new behaviour is required from some of the ports involved in the reconfiguration. The H.263 encoder in the final configuration needs to receive raw video frames from the video source component. However, this component is already delivering video to the existing MPEG encoder and must continue to do so until the end of the integration stage. The solution in this case is to have the output port of the video source deliver data to the input ports of both encoders until it disconnects from the MPEG encoder during integration—this is the purpose of the additional \texttt{echoTo} action associated with the video source output port in Figure 6.13.

Clearly this technique will not be useful in all situations; it can only be applied to divergent regions of the final path configuration that are not already running, and then only within available resources. However, even given these caveats a lot of common reconfigurations are covered: replacing one source or sink with another (changing “channel” or moving to a new location in a “follow-me video” situation), or re-routing of stream as in the example.
Figure 6.13: Setup/integration actions with adjusted stage boundary.
The main cause of smoothness violations under the basic algorithm is the execution time of the integration actions on each reconfigured path region. In our example reconfiguration, the time taken to shut down the existing MPEG codec and network endpoints is likely to be at least an order of magnitude less than starting up the H.263 components of the final configuration. If the integration of these two regions is started simultaneously—as it will be under the basic algorithm—there will be an undesirable interval between the display of the last MPEG frame and first H.263 frame. The obvious solution is to begin the integration of the final path region \( l \) seconds earlier than that of the initial path region, where:

\[
l = \text{latency}_{\text{final}} - \text{latency}_{\text{initial}}
\]

This will ensure that there is no gap—or, in the case that the magnitude of the latencies is reversed, overlap—between the two configurations.\(^4\) The latency values are computed simply by totalling the time taken by each object on the reconfigured region to execute all of its integration actions and forward the first data item to its downstream neighbour. The same calculation can be applied anywhere along the path that two parallel divergent regions occur, although it should be noted that there is little to be gained from integrating a region until all of the regions upstream from it have also been integrated. Resource shortages during the integration stage may be exacerbated by the use of this technique, if it results in the initial and final configuration running in parallel for longer than they otherwise would have. This may be an acceptable cost if other more noticeable smoothness violations are prevented; exactly where the line should be drawn is an application-specific decision. Figure 6.13 shows the example reconfiguration with an additional DELAY action added to the MPEG encoder component; this will effectively stall the integration of this path region for the specified time.

A further scheduling problem that can occur relates to the continuous—as opposed to integration—latency of the reconfigured regions; that is, the end-to-end latency of a media element passing through the object in a region when they are running in a static configuration. Suppose that instead of switching from MPEG to H.263 encoding, the example configuration is switching from MPEG to uncompressed, raw video. The end-to-end latency of the MPEG path might be, say, five frame times, while the latency of the uncompressed path will be much less, perhaps only one frame time. If the switch is made using the scheduling optimisation just discussed, the MPEG region will be shut down after a delay of five frame times, the uncompressed region started up. Assuming that the last MPEG data transmitted is frame number \( N \), the first uncompressed data seen will be frame \( N + 5 \); the intervening four frames are never seen! If the scheduling is not optimised and both regions are integrated simultaneously, a similar situation occurs, except that earlier frames are

\(^4\)It may be desirable to begin integration of the final path region in the example one "frame time" later than the equation indicates, so that the first H.263 frame arrives at the display \( \frac{1}{2} \)th of a second after the last MPEG frame, for a 15fps stream. However, not every stream type has such a consistent idea of frame time, and the general principle remains the same in any case.
Figure 6.14: Setup/integration actions with optimized integration scheduling.
lost.

This problem only becomes significant if the initial and final configurations carry data from the same stream, and the number of frames lost is enough to constitute a violation of temporal integrity. A potential solution involves adding additional delay to the region with the smaller latency—in this case an extra four frame-time delay buffer on the final path configuration could eliminate the frame loss entirely. Unfortunately, use of this technique implies that the end-to-end latency of the path can only ever increase or remain the same, but never decrease. However, the extra delay could be removed gradually, after the reconfiguration has completed, by playing the data out of the delay buffer slightly faster than the nominal data rate of the stream, until the buffer was empty—it could then be removed and discarded. While there are obvious questions of temporal integrity here, the idea of further “tweaking” a configuration after the atomic action has committed falls outside the scope of this thesis and will not be discussed further here.

6.2.3 Extension to multiple paths

The discussion so far in this chapter has dealt with reconfigurations of a single path. Many real-world reconfigurations will be far more complex and require multiple paths to be reconfigured simultaneously, with their temporal integrity intact.

Multiple paths may only become an issue when they are related; that is, when there is some interaction or dependence between them. If two (or more) paths are entirely independent, the algorithms presented above should be able to deal with reconfiguring them simultaneously, even within the same atomic action, by computing and executing a separate schedule for each path. The most desirable outcome would be to have the reconfiguration of all the paths proceed in parallel; resource constraints may make this impossible, forcing the reconfigurations to be serialised.

The situation is not so straightforward when path dependencies exist. Consider an the remote surveillance application, sending synchronised audio and video streams over separate connections. If a user switches to watching a different pair of streams, it is expected that the audio and video will switch simultaneously at the output endpoint, and that they will remain synchronised throughout. Other examples abound: a component may depend on data generated by another path, or two components may have to coordinate changes to shared attributes of some device. DJINN’s QoS subsystem is able to comprehensively model QoS dependencies between media streams and the components that process them; for instance, it is possible to specify that two streams must share the same frame rate, or that only two out of a set of five streams may be allowed to run concurrently. It would be reasonable to extend this mechanism to model reconfiguration dependencies between paths; the difficulty at present is that it is not clear exactly what need to be modelled.

Ad-hoc solutions are certainly possible for some simple cases, such as switching synchronised audio and video streams. In this scenario, the additional requirements are that the first new audio and
video data are received at their respective destinations within some small interval of each other, and that those first elements are correctly synchronised—that is, they come from the same point in the combined stream. Achieving this is a matter of adjusting the scheduling of the audio path reconfiguration with respect to the video path, to ensure that both reconfigurations complete at the same time. It might also be necessary to insert extra delay components (see page 167) to maintain synchronisation, if the end-to-end latencies of the audio and video configurations are very different.

Clearly simple schemes like this will not suffice for applications with hundreds of streams and more interesting dependencies. Further research is required in two areas: firstly, modelling of path and reconfiguration dependencies and, secondly, how to utilise those dependencies in the scheduling of actual reconfigurations.

6.3 Implementation

The DJINN framework includes a prototype implementation of scheduled reconfiguration, using the algorithms presented above. The current implementation couples a modified model-active control interface with the existing DJINN event handling system to provide a straightforward mechanism for managing the execution of active layer operations. While the discussion of scheduling algorithms in the earlier sections of this chapter indicates that applications can exercise some control over the quality/resource/time tradeoff of individual reconfigurations, there is at present no interface to specify the necessary parameters; it is necessary to recompile the atomic action code to change the scheduling behaviour. However, there is nothing preventing such an interface from being added later.

6.3.1 Managing the active layer update

Most of the reconfiguration scheduling process runs in the model, as part of committing the enclosing atomic action. The active layer is only involved at the very end of the procedure, when the setup and integration actions are finally executed. A key aspect of the implementation is determining firstly, how to inform the active layer of what is to be done and, secondly, ensuring that it happens in the right order with the appropriate timing.

Three distinct approaches to managing the final active layer update were considered. All assume that the reconfiguration has been completed to the point of computing both the setup/integration boundary for every reconfigured object, and the relative timings for the integration of each reconfigured region.

- **Model-driven.** In this scheme, the model explicitly executes each outstanding setup or integration action by making a remote invocation on the appropriate active object.
Implementation

- Active-layer agent. The opposite of the first approach, here the model "downloads" the schedule and list of pending actions to one or more "smart agents" co-located with the active layer objects. The agents are responsible for invoking the actions according to the schedule.

- Hybrid solutions. This technique seeks to keep as much processing as possible in the model, while utilising the real-time properties of the active layer to ensure that the schedule is met.

The obvious drawback of the model-driven scheme is that it requires the model to perform an arbitrary series of remote invocations on the active layer in real time. This can only be reliably achieved if the host(s) running the model, and the network links between the model and active layers, can offer similar real-time guarantees to the active layer itself—an unlikely situation in practice. Driving the entire update from the active layer appears to be a more promising approach, but has the potential disadvantage that it adds complexity to the active layer that may be difficult or impossible to support on small or under-resourced devices. Consequently, for the current prototype implementation at least, I have chosen to use a hybrid solution that splits the work between the application layers.

The scheduler takes advantage of DJNN's existing ability to send events along a path, interleaved with media data. This function is used to deliver events indicating to each object along the path that the setup or integration actions associated with a particular reconfiguration should be executed; once all of the relevant actions are complete the triggering event is forwarded downstream to the next element of the path. The actions themselves are sent in advance by the model to each reconfigured active object and queued until the appropriate event is received.

6.3.2 Model→active interface

Supporting this update scheduling mechanism requires a fundamental change to the interface between the model and active layers. The conventional approach, described in Chapter 4, has atomic model objects directly invoking remote operations on their active layer counterparts. In the current implementation, this familiar interface is replaced by a single remotely-invocable method, deliver, that allows the model to send an event to an individual active object. For reconfiguration purposes, a delivered event will either:

- Encapsulate the name and arguments of a method to be invoked on the active object, along with a "key" to trigger that invocation. The event will be queued by the active object until the key is received.

- Trigger the execution of all queued events associated with a particular key. The event will be forwarded downstream where it may trigger further invocations, passing transparently through any path members that do not require reconfiguration.
Each reconfiguration will generate unique keys for its setup and integration stages, allowing these to be triggered separately.

Using this approach, the model needs only to ensure that appropriate trigger events are delivered to the upstream endpoint(s) of the initial (and final, if they are different) path configurations. The event forwarding mechanism will automatically trigger the reconfiguration of all downstream objects in the correct order.

6.3.3 A complete example

The behaviour of the scheduling implementation is best appreciated by an example. Figures 6.15–6.20 illustrate the integration of the example MPEG→H.263 reconfiguration used throughout this chapter. The schedule has been computed using the latency optimisation presented on page 165; I have assumed that insufficient resources are available to adjust the setup/integration boundary of any objects away from the default. Figure 6.15 shows the state of the active layer immediately after the setup stage actions have been executed (these actions are shown in light grey text on the diagram). The new components have been created and connected to each other, but media data is still flowing through the initial MPEG configuration. For simplicity, all timing data are shown in terms of frame times with respect to the video stream, at whatever rate it is running. As the labelling on the media elements shows, the end-to-end latency of the MPEG path region, from the input port of the encoder to the output port of the decoder, is five frame times.

One frame time later, the integration stage is triggered by the model delivering an event to the output port of the video source (recall that this port is the most upstream object in the first reconfigured region on both path configurations). The event will contain the key for the integration stage of this reconfiguration, and indicates that all actions associated with that key should now be executed. Therefore, the port will perform its queued integration actions before forwarding the event downstream, which now means towards the new H.263 encoder. This situation is shown in Figure 6.16, with the next video frame following the event towards the encoder. Note that the MPEG components are still running, so data continues to be delivered to the display.

Figure 6.17 shows the state of the active objects after a further frame time has elapsed. The H.263 encoder has started up, but since it has only seen one incoming video frame so far (element N + 6) it has yet to produce any output. The trigger event continues to move downstream and another frame is produced by the MPEG decoder.

We now move forward two more frame times, to the point where the schedule dictates that the old MPEG components can be shut down. Earlier discussion showed how these actions could be held until the correct time by the use of an extra DELAY action in the integrate stage of the MPEG encoder. This example presents an alternative approach that achieves the same behaviour. Integration of the MPEG path region (region B in Figure 6.11) is started by receipt of a second event from the
Figure 6.15: Executing integration stage, $t = 0$. 
Figure 6.16: Executing integration stage, $t = 1$. 
Figure 6.17: Executing integration stage, $t = 2$. 
model, delivered directly to the disconnected input port of the MPEG encoder. By the time this event arrives (see Figure 6.18, the first event has passed across the GSM network and almost reached the H.263 decoder, the H.263 encoder has produced its first frame, and the MPEG decoder is still managing to deliver data to the display.

After a further frame time, the second trigger event has reached the MPEG decoder (Figure 6.19); all objects prior to this point have been shut down and deleted. The decoder itself has just produced the last item of data it has available—the same media element that entered the encoder in Figure 6.15. Meanwhile, the first event has passed through the H.263 decoder and is on its way towards the display. The situation here is slightly unusual, in that we apparently have an input port with two output ports connected to it. Consistency checks in the model would prevent this from occurring in a static configuration; the active port itself does not care what is actually connected to it and will happily accept data from any other object. Fortunately this is only a transient anomaly, as the MPEG decoder output port will disconnect itself as soon as the second trigger event reaches it.

Figure 6.20 shows the final state of the active layer, after the integration phase has completed. The display is now receiving frames from the H.263 codec, while the MPEG components have been deleted. As for the trigger events, they will have been discarded by the display input port, since there were no further downstream objects to be reconfigured.

6.4 Summary

The final phase of a Djinn reconfiguration is the updating of active layer objects to match the new configuration of the application model. This transition must often be performed while the active objects are running, without breaking the temporal or data integrity of the object involved, or adversely affecting the operation of any other stream. Because the objects may be processing media data while the transition takes place, there is generally only one chance to get it right—any mistakes will be visible to users of the application. This chapter has presented algorithms for ordering and scheduling the updating of active objects to meet these demands.

Section 6.1 discussed the requirements of the active layer update process in detail and motivated the need for scheduling. This section introduced the idea of smoothness and the notion of basing the update schedule on maintaining the end-to-end properties of paths rather than individual objects.

Algorithms for computing and executing the schedule were presented in Section 6.2. The outstanding reconfiguration actions for each active object are divided into two stages: a setup stage that can be performed without affecting the current operation of the reconfigured path, and an integration

---

Footnote: This is the technique used by the implementation. Its effect is equivalent to the earlier approach; it simply proved easier to implement things this way at the time. The current scheduler actually generates different setup and integration keys for each reconfigured region. These may be "chained" together, so that reconfiguration flows from one region to another—as with regions A and C in the example—or not, such that another event must be delivered to start reconfiguration on the region.
Figure 6.18: Executing integration stage, $t = 4$. 

[Diagram with detailed nodes and connections]
Figure 6.19: Executing integration stage, $t = 5$. 

...
Figure 6.20: Executing integration stage, $t = 6$. 
stage that completes the transition from initial to final configuration. The aim is to move as much of the update as possible into the setup stage, allowing the final configuration to be integrated quickly and with minimal loss of data or temporal integrity. Several optimisations to the basic algorithm are discussed.

Section 6.3 briefly addressed the implementation of these scheduling algorithms in the DJINN framework. The implementation uses the existing DJINN event-forwarding mechanism to schedule the actions within each reconfigured path region automatically. This section includes a detailed example showing how the algorithms behave with a typical reconfiguration adapted from the remote surveillance case study.
CHAPTER 7

EVALUATION

The previous three chapters have presented an architecture for composing, controlling and reconfiguring distributed multimedia systems. In particular, Chapters 5 and 6 discussed mechanisms for structuring and executing reconfigurations on active continuous media streams, that respect the temporal properties of those streams and maintain the consistency of the enclosing application. A prototype implementation of the architecture exists as the DJINN framework; details of the implementation can be found in the earlier chapters.

This chapter presents the results of an experimental evaluation of DJINN; the aim being that these results should provide some validation of the underlying architectural concepts. Neither the architecture nor the DJINN prototype are sufficiently well-developed as yet to attempt any formal analysis and validation of their properties. However, it is hoped that the results given here will show that the architecture offers a viable approach to the deployment of at least certain classes of distributed multimedia application.

The remainder of this chapter is organised as follows:

- Section 7.1 first summarises the current state of the DJINN prototype, indicating which of the architectural features discussed previously are implemented and which are not. The rest of the section presents experimental methodology: the structure of the test applications and reconfigurations, the runtime environment, and details of the experiments performed.

- Section 7.2 presents an overview of the results of the experimental trials, as well as some analysis and discussion of the results and their implications for the design of the architecture and framework.
7.1 Methodology

The experiments were designed to test the DJINN prototype in a "real world" application scenario based on the remote surveillance application introduced in Chapter 3. In order to generate useful data from the experiments, new instrumentation code was added to the model and active layers. Each active object keeps a timestamped log of every action it performs as well as each incoming and outgoing media data element or event. Similarly, the atomic action code logs the progress of every reconfiguration performed on the application model. Log data is dumped to files, allowing it to be stored and analysed offline.

7.1.1 Current framework implementation

As discussed previously, the model layer of the DJINN prototype is implemented entirely in Java, with the exception of a single combined Java/C module that handles the saving and restoring of object state in aborted atomic actions. The model layer code is expected to be portable to any platform with a JDK-1.1 compliant Java virtual machine and an ANSI C compiler capable of producing shared libraries. It has been successfully compiled and executed on machines running the Microsoft Windows, Sun Solaris, FreeBSD and Linux operating systems.

The active layer contains a considerable amount of C/C++ code, both for driving specific media-handling components and for general tasks such as object serialisation and low-level buffer handling. For the most part, the current DJINN active layer will run on the same platforms as the model layer; however, certain components require the presence of particular hardware or operating system facilities. For example, the framework includes a generic video-capture component that requires Microsoft's "Video for Windows" API (and thus the entire Microsoft Windows operating system) in order to work. Likewise, many of the video codec components will run on the various UNIX-like operating systems (Solaris, FreeBSD, Linux) but not under Windows.

With regard to the architectural features and mechanisms presented in the preceding chapters, everything is implemented in the current DJINN prototype with the exception of:

- **QoS support.** The model layer does not include descriptions of component's QoS characteristics. Likewise, there is currently no admission-control mechanism or resource management in the active layer.

- **Real-time support.** None of the operating systems supported by the prototype offer true real-time scheduling or reservation-based access to system resources. Thus, applications operate in a best-effort mode.

- **Multiple paths.** Simultaneous reconfiguration of multiple path objects is not supported by the reconfiguration scheduling algorithms.
• **Parameterised reconfigurations.** Reconfigurations as described in Chapters 5 and 6 are parameterised in terms of an application's desired smoothness, resource usage or integration time. The current prototype has no means to specify these parameters on a per-reconfiguration basis; changing the behaviour of the reconfiguration scheduler involves modifying and recompiling the atomic action code.

• **Per-component integrity checks.** Support for component- and class-specific structural integrity tests exists but no such tests have been implemented as yet. Various generic tests are performed, mostly to ensure that connections can only be made between compatible input and output ports.

A second DJINN project prototype, developed by Hani Naguib, includes support for QoS modelling and resource management and runs on the Chorus real-time operating system. However, this version of DJINN does not include any of the reconfiguration mechanisms proposed by this dissertation. Unfortunately the two prototypes have diverged too far to be easily merged, although there is a long-term plan to develop an integrated DJINN platform that integrates both sets of technologies.

### 7.1.2 Test application design

All of the experiments were carried out on the same test application, a simplified version of the remote surveillance system. The general structure of this application is shown in Figure 7.1. Note that, with respect to the original remote surveillance system—see Section 3.1.1 (page 51) and Figure 4.27 (page 104)—the test application lacks the central video server host. Instead, compressed video is broadcast directly from the various sources, which in this case are files or pre-recorded video rather than live cameras.

Three different reconfigurations were implemented, beyond the obvious actions of adding, removing, starting or stopping source streams or clients. The effects of each reconfiguration are illustrated by Figures 7.2–7.4:

**Change source.** An existing client switches from viewing source A to viewing source B. Network endpoints and transcoding components will be created as necessary—for example, if the client is using M-JPEG encoding and source B had no other M-JPEG clients before the reconfiguration. Likewise, if the client was the only M-JPEG client of source A, A's MPEG → M-JPEG transcoder and M-JPEG source endpoint will be deleted.

**Switch encoding.** The selected client continues to view the same stream, but switches to an alternative encoding scheme; that is, if it is viewing the MPEG stream from source A it will switch to
Figure 7.1: Structure of the experimental test application.
the M-JPEG encoding of the same stream, and vice-versa. In the real surveillance application, the encoding change would generally be associated with a network hand-over, for example from the fixed Ethernet to WaveLAN. As with changing sources, transcoding and network endpoint components may need to be created or deleted.

**Move client.** A client is moved to a different host, equivalent to deleting the existing client and creating an identical one in the new location. This reconfiguration would be used in a "follow-me video" situation, where a user is moving between multiple fixed terminals at wishes to be able to view a selected stream at his current location.

There are number of problems with the MPEG codec used in the current prototype that limit the range of arbitrary reconfigurations that can be performed. The codec components are based on the Berkeley MPEG-1 encoder and decoder [RPSL94]. Both of these were originally built as monolithic, standalone programs, reading input data from files and writing their output to other files (or displaying decoded frames directly). The code uses a considerable amount of global data and as such has not adapted well to life as a shared library in the component-based DJINN environment. The upshot of this is that two or more MPEG encoders or decoders cannot be run simultaneously within the same Java virtual machine, due to their need to access and modify the same global data. It
is possible to create multiple encoder or decoder instances as long as only one is running at once, although such configurations are not reliable over the long term.

Thus, certain reconfigurations, such as switching the encoding of a client from MPEG to M-JPEG then back to MPEG, should not be performed as they may cause the DJINN runtime to hang or crash. It should be emphasised that this is a problem with the C code of the codec components, not DJINN itself, which could be solved by replacing the offending code with new codec, written to operate correctly as a shared library. Unfortunately suitable alternative open-source code was not available at the time the framework was being implemented. The M-JPEG code used in the test application is based on code from the Independent JPEG Group\(^1\) and does not suffer from any such problems.

7.1.3 Experimental design

Two sets of experiments were run, to test firstly the “basic” scheduling algorithm described in Section 6.2.2 (page 161); and secondly the optimised version of the algorithm that takes account of the latency of individual active layer actions (Section 6.2.2, page 164). Latency figures derived from the result of the first experiment set were fed back into the model to improve the performance of the reconfigurations in the second set.

The first experiment set consisted of the following five experiments:

1. **Baseline latency and jitter.** The test application was used to play a single video stream on a single client without any reconfiguration beyond the initial setup of the application. The aim of this experiment was to obtain baseline data for the end-to-end performance of the system in a static configuration, for comparison with data from the later reconfiguration experiments.

   The experiment was run with both MPEG and M-JPEG clients and with four different arrangements of source and client hosts, for a total of eight trials. In every case the same MPEG

\(^1\)http://www.iijg.org/
source stream was used and the application run until 5000 frames had been received and displayed by the client.

2. **Non-structural reconfiguration.** A simple variation of the first experiment, designed to test and verify the effect of basic non-structural reconfigurations on end-to-end performance. The client was stopped and later restarted a total of 10 times over the course of each trial, with at least a 10 second interval of continuous running before each stop reconfiguration and at least three seconds before the subsequent start operation. This experiment used the same source stream and sequence of eight trials as the previous, except that each trial was run for a total of 1000 frames, this being sufficient to cover the 10 start/stop reconfigurations.

3. **Change source configuration.** A single client was switched repeatedly between two different source streams. The source streams had identical frame size, frame rate and MPEG coding parameters, although they were of different lengths. The client was switched from stream A to stream B then back to stream A; this pattern was repeated 10 times with an interval of 10 seconds between each switch. Two trials were run with different configurations of source and client hosts.

4. **Switch encoding reconfiguration.** The encoding of a single client was switched repeatedly from MPEG to M-JPEG and back to MPEG. The client was viewing the same source stream throughout. A total of 10 MPEG→M-JPEG→MPEG trials were performed, with a 20 second interval between each switch.

5. **Move client reconfiguration.** A single client was moved between two hosts while displaying a video stream sourced on a third host. Four trials were run, to test both MPEG and M-JPEG clients with two different configurations of source and sink hosts. Each trial involved 10 repetitions of a host A → host B → host A switching sequence, with an interval of 10 (for the MPEG trials) or 20 (for the M-JPEG trials) seconds between each reconfiguration.

The second set of experiments repeated the final three from the first set, using the optimised reconfiguration scheduling algorithm. The baseline and non-structural experiments were not run again using this algorithm as there is no benefit to be gained from its optimisations in those cases; the baseline scenario does not involve any reconfiguration while both the basic and optimised versions of the algorithm generate the same schedule for the non-structural start/stop reconfiguration.

All of the experimental trials used the same two MPEG-encoded source streams. Both streams use the same MPEG encoding parameters and consist of, respectively, 330 and 190 full colour 304x224 frames to be run at 15 frames/second.

The test application was built as a DJINN composite component, with the reconfigurations implemented as methods of the new component class. A small program was written to create and manage instances of the test component class; this program also contains methods to execute each of the
Figure 7.5: Test application screenshot.

Experimental trials described above on a set of hosts supplied through command line arguments. The test program includes a simple graphical user interface allowing source streams and clients to be set up, and new reconfiguration scenarios tested. Figure 7.5 shows this interface along with two client windows displaying the streams used in the experiments.

Execution environment

The experiments were run on a network of Intel i686-class (Pentium II and III) machines running the RedHat Linux\textsuperscript{2} 6.0 or 6.1 operating system (all with Linux kernel 2.2.5) and the Blackdown\textsuperscript{3} port of Sun's JDK 1.1.7. All machines were connected to a switched 100Mbps Ethernet. Apart from essential system daemons and the X-Windows server and window manager, no other software was active on the machines running the experiments. System load before each run, as measured by the Linux uptime command, was always less than 0.05. The network load imposed by these experiments was quite low: the typical size of a compressed video frame from the streams used in the experiments was in the range of 4--16Kbytes. Even with two streams running at 15frames/second the total data transferred is less than 0.5Mbytes/second—i.e. less than 5% of the capacity of each network link. Given that under "normal" load conditions, the Ethernet switch was able to provide

\textsuperscript{2}http://www.redhat.com/
\textsuperscript{3}http://www.blackdown.org/
effectively a dedicated 100Mbps link between any pair of hosts on the network, it is unlikely that the results will exhibit any observable perturbation due to network queuing. Indeed, no video frames were dropped because of network congestion at any point in the trials.

The timestamps on log entries generated by DJINN are taken from the system clock with a resolution of 1ms. System clocks on the test network were synchronised against a central master clock via NTP [Mi91]. NTP will allow some clock drift against the master clock; in practice however this rarely exceeded 1ms on any machine over the course of the experiments. Where clock drift—rounded to the nearest millisecond—would have had an effect on results, all affected timestamps were corrected to bring the drifting clock effectively back into synchronisation with the master.

7.2 Results and discussion

The figures and tables in this section present a representative sample of results from the experimental trials described above. Most of the experiments involved multiple trials on different combinations of host machines; each set of results shown here is taken from a single such trial. The behaviour of the system across the remaining trials was generally similar in all cases. In the interests of clarity and avoiding repetition, this chapter does not always include all of the results for each trial that is discussed; the full results can be found in Appendix A.

7.2.1 Non-optimised trials

This section covers the experiments run using the standard non-optimised reconfiguration scheduler. Five sets of trials were conducted as described above.

Baseline operation

Figures 7.6–7.8 show the behaviour of the test application in continuous operation, playing an MPEG encoded video stream, respectively. The figures show the media rate (equal to the frame rate in these trials), end-to-end latency and end-to-end jitter along the video source→display path, over the entire duration of the trial as well as more detailed views over a 30-second window. The interval used in the 30-second views was chosen to show an instance of the video stream “rewinding” before looping back to the first frame again. A smaller set of results for the same configuration playing an M-JPEG stream is given in Figure 7.5; the observed behaviour of the system in the M-JPEG scenario was otherwise similar to the MPEG case.

The media rate is calculated with respect to any other media elements received within a one second time window leading up to the receipt of each element. Likewise, the end-to-end latency shown in the figures is a one second rolling average using the same sliding window. The jitter at each point is calculated relative to this rolling average.
Figure 7.6: Media rate for baseline operation with MPEG stream.
Figure 7.7: End-to-end latency for baseline operation with MPEG stream.
Figure 7.8: End-to-end jitter for baseline operation with MPEG stream.
(a) Media rate over 30s interval.

(b) End-to-end jitter 30s interval.

Figure 7.9: Baseline operation with M-JPEG stream.
A number of points are obvious from examination of these figures. Firstly, the application has clearly been unable to maintain true "real-time" performance throughout the trial, as there are several visible glitches that cannot, in this situation, have been caused by reconfiguration activity. In itself this is not terribly surprising, given that the application was both written in Java and running on a non-deterministic OS and network platform that offered no QoS guarantees. It is gratifying to note that despite these limitations the application was able to maintain a relatively constant frame rate between reconfigurations, for both MPEG and MJPEG streams (see Figures 7.6b and 7.9b), with correspondingly low jitter. The periodic pauses in output visible on the graphs are caused by the source MPEG stream "rewinding" back to the first frame; the MPEG decoder requires in the region of 1-1.5s to resynchronise and produce a new output frame after receiving out-of-sequence data from the start of the stream.

Doubtless the reader has already noticed that the act of rewinding an MPEG stream appears to cause a serious increase in the end-to-end latency of the application path. This is another unfortunate side-effect of the MPEG decoder implementation used in the current DJINN prototype. When the decoder produces an output frame, it does not indicate which input frame the decoded data corresponds to. The enclosing DJINN component maintains a FIFO list of frame numbers that have gone into the decoder and assigns these to output frames in the same order. In normal operation, this scheme performs adequately, even given the fact that the frames in an MPEG stream are not necessarily encoded in strict presentation order. However, when the stream is rewound, the decoder will not produce any output until it has fully resynchronised—any input data received prior to this point will effectively be dropped. Thus the component will assign an "old" frame number to the next output frame, giving it the appearance of having a much higher latency than it has in fact experienced. This mismatch between actual and observed frame numbers will increase every time the stream rewinds. Careful examination of the graphs will show that the indicated increase in latency is proportional to the time taken for the decoder to resynchronise before producing a new output frame.

The periodic negative spikes in the jitter graph are also associated with stream rewinding. These glitches are in fact caused by the MPEG stream player component at the upstream end of the video path. This component is scheduled periodically and is supposed to produce the next encoded frame of the MPEG stream each time it runs. However, when the component reaches the end of its input file, it does not produce any data in that period; the stream is rewound and the first frame will be sent again the next time the player is scheduled. Thus, there is a double-length interval between the last frame of the stream and the replaying of the first frame. Further downstream, the MPEG decoder produces frames at a constant rate. The first output frame after the stream rewinds will have lower end-to-end latency than the frames on either side of it, due to the decoder effectively absorbing half of the two frame-time interval before the frame. Hence, the jitter briefly spikes downwards, but

---

4Specifically, an increase in the delay through the MPEG decoder, which contributes the bulk of the end-to-end delay in all of the experiments.
returns close to zero immediately afterwards.

It must be stressed that the end-to-end latency of the path does not in fact increase when the stream rewinds; the perceived increase is a result of dropped frames not being detected correctly. It would be relatively easy to correct the latency data for this effect after the fact, to make the graphs look “right”, but much more difficult to do this while the application is running, since the time taken to resynchronise (and thus the number of frames dropped) depends on the encoding properties of the stream and cannot necessarily be predicted in advance. I have chosen not to attempt any corrections to the graphs presented here, in the interests of minimising sources of error in the results and to give an honest evaluation of the DJINN prototype’s performance. In any case, it is the change in latency during reconfigurations that is of greater interest here, rather than the absolute value at some point in time. A preferable solution would, of course, be to use a decoder that provides the required frame number data.

The graphs show a number of further glitches that do not correlate with any rewinding of the source stream. In the non real-time environment used for the experiments, there are several possible causes of such QoS violations:

- Network congestion leading to late delivery of data.
- Activity of other processes on one of the hosts, denying the DJINN runtime immediate access to the CPU.
- Java garbage collection. All other Java threads are suspended while the garbage collector runs.
- Contention for I/O devices such as the disk or display.
- Virtual memory paging activity.

It is difficult to say with any certainty which of these might have been responsible for any given glitch, although in a Java environment with a standard JDK, garbage collection is always a likely culprit.

Non-structural reconfiguration

Figures 7.10 and 7.11 show the test application undergoing non-structural reconfiguration—stopping and restarting the client components—while playing MPEG and M-JPEG encoded versions respectively of the same stream. These graphs show end-to-end latency over the whole trial and in more detail over a single pair of stop/start reconfigurations for each encoding type; the times at which the setup and integration stages of each reconfiguration were initiated by the application model and completed by the active layer are indicated on the detail views.
Figure 7.10: Non-structural reconfiguration with MPEG stream.
Figure 7.11: Non-structural reconfiguration with M-JPEG stream.
The MPEG results in particular clearly illustrate the effects of buffering in the MPEG decoder and associated DJINN component. The component code provides a buffer with enough capacity for several seconds worth of encoded video—the exact amount depends on the frame rate—on the input port of the decoder. In addition, the decoder may itself buffer data internally, such as when frames must be re-ordered prior to playback. When a video client in the test application is stopped, it stops immediately; any buffered data will be stalled until the client restarts. Meanwhile, the source will continue to run, multicasting to any other clients it might have. Thus, when the client restarts it will first flush out the buffered data, which will have its apparent latency increased by the length of time that the client was stopped. This can be seen in Figure 7.10a, where a large increase in measured latency is observed immediately after each reconfiguration. Eventually new data reaches the decoder and emerges with a more reasonable latency, although greater than it should be after having had to wait for the old data to be flushed out. Measured latency is also affected by rewinding of the source stream, as in the baseline trials.

In the case of non-structural reconfiguration with an M-JPEG stream (Figure 7.11), the MPEG decoder is part of the source rather than the client and remains running throughout. Therefore, measured latency will be affected only by the rewind problem; there is never any "old" data to be flushed out of the decoder. This can be clearly seen in Figure 7.11b, with the latency remaining constant either side of the reconfiguration. The occasional transient peaks in the M-JPEG latency plots occur when the client happened to be stopped while it was decoding a media element. Such elements are stalled until the client is restarted and thus have their latency increased by three seconds.

Otherwise, the application behaves very similarly in this experiment and the baseline scenario discussed above. The reconstructions themselves complete in less than 100ms ($t_{\text{integrate}}$ is typically in the region of 50ms; that is, slightly less than one frame time in these trials) so appear to happen practically instantaneously.

"Change source" reconfiguration

Results for the first of the structural reconstructions (switching source streams with a single client) are shown in Figure 7.12, for MPEG encoded streams only. The figures give a detailed view of the end-to-end latency and jitter of the video path over a single reconfiguration. The graphs also indicate various milestones in the execution of the reconfiguration:

- Initiation of the setup and integrate stages of the reconfiguration, by the model delivering a trigger event to the upstream endpoint of the path.

- Completion of processing actions in the setup and integrate stages, measured at the downstream endpoint of the path.
Figure 7.12: "Change source" reconfiguration with MPEG streams.
As the graphs indicate, the reconfiguration itself completes in a little more than 200ms, with the integrate stage consuming slightly more than half of this time. By the time the reconfiguration has completed, a new multicast sink has been created (and the old one deleted), and the MPEG decoder is receiving data from the new stream. However, following the reconfiguration a full 3.8s appears to elapse before any data from the second stream is received by the display. Depending upon the smoothness parameters specified for the reconfiguration, this may constitute a violation of temporal integrity, even though the actual switch between the two streams is very smooth, completing in less than one frame time.

The cause of the delay is twofold. Firstly, as with the non-structural reconfiguration, the MPEG decoder needs to flush out any old data from the first stream before it can begin decoding the frames from the second stream. If the stream player at the source end of the stream is delivering frames even slightly faster than the decoder can consume them—given that the timing code for the stream player is written entirely in Java, this is not entirely unlikely—there could be a significant amount of buffered data\(^5\) to be flushed. Secondly, the reconfiguration shown in the figures happens to have occurred just as the MPEG decoder is rewinding back to the start of the first stream. There is therefore the usual delay while the decoder resynchronises, accompanied by an equal increase in measured—not actual—latency, by 1.22s in this case. Throughout the resynchronisation period the decoder is silently dropping frames, although their identifiers are still (erroneously) placed on the FIFO queue of pending output frames.

Thus, by the time the decoder generates the next output frame, there will be an excess of frame identifiers from the first stream in the queue, which will be attached to the data emerging from the decoder. Some of this data—up to 1.22s worth in the worst case—may in fact belong to the second stream, so the switch between first and second streams may have taken place earlier than shown on the graph. Taking this into account, the switching delay, including the time taken for the decoder to resynchronise, could be as low as 3.8 - 1.22 = 2.58s. That figure is close to the average latency of the decoder component and a reasonable estimate of the time that intuitively should elapse, between adding the first frame from the second stream to the input queue of the decoder and its decoded equivalent emerging from the output. This analysis matches both the qualitative observations of the application display made while this experiment was running, and inspection of the log entries generated by components upstream from the MPEG decoder.

"Switch encodings" reconfiguration

Results for the "switch encodings" reconfiguration are presented in Figures 7.13 and 7.14 for the M-JPEG→MPEG and MPEG→M-JPEG transitions, respectively. As with the previous plots, the

\(^5\)The current DJinn component wrapper around the MPEG decoder does not provide any frame-by-frame data on buffer occupancy; it does however indicate when the buffer has under- or over flowed. This is an obvious shortcoming of this component which will be corrected in any future revisions of the software.
timing of the key reconfiguration events is indicated, including receipt of the last frame of the initial configuration and the first frame of the final configuration.

The major difference between the two transitions is the respective delay before the first frame from the "new" encoding is received. In the M-JPEG→MPEG case this is 2200ms whereas for the MPEG→M-JPEG transition it is only 232ms—an order of magnitude difference. This is a result of the relatively enormous latency of the MPEG decoder. Application performance is not helped in this instance by the unfortunate coincidence of a source stream rewind at the same time as the reconfiguration, with the pause in output making the transition appear to take much longer than it actually has. The relatively high jitter immediately after the transition can probably be attributed to the load imposed on the DJJNN runtime by the reconfiguration: starting an MPEG decoder is a heavyweight operation which is likely to trigger virtual memory activity and garbage collection. In contrast, the MPEG→M-JPEG transition is much smoother, with a break in output of less than three frame times.

The other point worth noting here is the frame numbers on either side of the transitions. For the M-JPEG→MPEG switch, the last M-JPEG frame is number 307 and the first MPEG frame is number 349. For the MPEG→M-JPEG reconfiguration the corresponding values are 618 and 626. The disparity in the first transition arises because the M-JPEG frames have already passed through both an MPEG decoder and an M-JPEG encoder; thus, their latency is much higher than the frames delivered directly to the newly created MPEG decoder in the client and a given M-JPEG encoded frame will arrive later than its MPEG encoded counterpart. However, by the time frame 349 is actually displayed by the MPEG client, the old M-JPEG client would have displayed as far as frame 340 had it been left running—so the actual discrepancy is only nine frames, or around 600ms. This can be accounted for by the fact that the MPEG decoder on the client host has a lower latency than the MPEG/M-JPEG transcoder combination at the source, both because the client has less work to do and because it happened to be running on a more powerful machine in this instance.

"Move client" reconfiguration

Finally, Figures 7.15 and 7.16 illustrate the "move client" reconfiguration for both MPEG and M-JPEG streams. The layout of the graphs is identical to that of the previous experiments.

The most obvious feature of the MPEG plots in Figure 7.15 is the reduction in end-to-end latency immediately after the reconfiguration. This can be explained firstly by the fact that the host decoding the stream in the final configuration is a dual-CPU machine and thus able to decode somewhat faster by, for example, not preempting the decoder thread in order to display a decoded frame. Secondly, the client in the final configuration was created, powered on and started running all within the same atomic action. The initial client, on the other hand, was powered up by one atomic action and started later by another, once the source stream was running. A client in the ON state has an open UDP socket, listening to the multicast group of the source stream. Any substantial delay between powering on and starting the client may result in a backlog of buffered data waiting to be read.
Figure 7.13: "Switch encodings" reconfiguration for M-JPEG→MPEG transition.
Figure 7.14: "Switch encodings" reconfiguration for MPEG→M-JPEG transition.
Figure 7.15: "Move client" reconfiguration with MPEG stream.
(a) End-to-end latency over single reconfiguration.

(b) End-to-end jitter over single reconfiguration.

Figure 7.16: "Move client" reconfiguration with M-JPEG stream.
from the socket. Clearly, this is what has happened here—the client in the final configuration has started up with little or no buffered data and therefore exhibits a lower latency in processing new data when it arrives.

The rapid increase in latency shown by the final client shortly after the reconfiguration can be attributed to its having started up halfway through the playback of the source stream. Figure 7.15a clearly shows two short pauses in the video output where the decoder has found it necessary to drop a small number of frames in order to correctly synchronise itself with the incoming data.

The MPEG plots also show the typical startup delay before the first decoded frame is displayed. In contrast, the M-JPEG client in Figure 7.16 is able to generate an output frame almost immediately after the reconfiguration has completed. This reconfiguration has occurred shortly before a rewind of the source stream, as witnessed by the pause in output and increase in latency at the $t = 50000$ ms point.

### 7.2.2 Active object behaviour

Tables 7.1–7.3 present a representative sample of the latency statistics gathered from individual active object classes. They show the time taken to execute methods of three different component types on three different hosts. The times are averaged across all invocations of each method on each host over the course of the experiments. The “startup latency”—the additional processing time required for the first media element after the component is started running—is zero for all of these component classes. All times are expressed in milliseconds.

For reference, host **spitfire** has dual Intel Pentium II CPUs running at 450MHz and 256MB of RAM. Host **tetley** is similar, but has dual 500MHz Pentium II processors. Host **tanglefoot** has a single Intel Pentium III CPU running at 600MHz and 256MB of RAM. This host ran the application model and GUI for all of the experiments; in some trials it was also used as a source or sink host for video data. All machines have identical Intel i82557 Fast Ethernet adapters.

These results do not show the host with the faster CPU as being particularly quicker at executing basic component operations—in fact, in many cases it appears to be slower than the other machines. An operation such as **poweron** is typically not CPU-bound, as it involves allocating memory, opening disk files and network sockets, etc. Having extra CPU cycles available will not have much effect on these activities. This observation is supported by the large standard deviations recorded for many of the operations. Often these are the result of a few very large latency values, one or more orders of magnitude larger than the “typical” values. This amount of variation can be expected when performing operations—such as opening a file—that will eventually require a system call into the

---

4While it is not strictly necessary to buffer incoming **UDP** data— **UDP** is after all an unreliable protocol—there is nothing to prevent a **TCP/IP** implementation from doing so if it wishes. Data may also be buffered within the network hardware. The client components could be modified to work around this behaviour by reading and discarding any buffered data at startup; this would reduce the apparent latency of the client in this situation.
<table>
<thead>
<tr>
<th>Host</th>
<th>spitfire</th>
<th>tanglefoot</th>
<th>teteleys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>Mean (ms)</td>
<td>St. Dev.</td>
<td>Mean (ms)</td>
</tr>
<tr>
<td>powerOn</td>
<td>13.8</td>
<td>8.46</td>
<td>30.3</td>
</tr>
<tr>
<td>powerOff</td>
<td>25.8</td>
<td>20.0</td>
<td>26.4</td>
</tr>
<tr>
<td>start</td>
<td>9.96</td>
<td>7.41</td>
<td>31.6</td>
</tr>
<tr>
<td>stop</td>
<td>11.7</td>
<td>18.7</td>
<td>14.0</td>
</tr>
<tr>
<td>setRate</td>
<td>0.611</td>
<td>0.494</td>
<td>2.83</td>
</tr>
<tr>
<td>setLoop</td>
<td>0.528</td>
<td>0.506</td>
<td>0.583</td>
</tr>
<tr>
<td>Media latency</td>
<td>2506.6</td>
<td>278.4</td>
<td>2362.9</td>
</tr>
</tbody>
</table>

Table 7.1: Latency results for MPEG decoder component.

<table>
<thead>
<tr>
<th>Host</th>
<th>spitfire</th>
<th>tanglefoot</th>
<th>teteleys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>Mean (ms)</td>
<td>St. Dev.</td>
<td>Mean (ms)</td>
</tr>
<tr>
<td>powerOn</td>
<td>14.5</td>
<td>18.8</td>
<td>19.8</td>
</tr>
<tr>
<td>powerOff</td>
<td>8.67</td>
<td>7.48</td>
<td>8.85</td>
</tr>
<tr>
<td>start</td>
<td>35.2</td>
<td>46.0</td>
<td>14.9</td>
</tr>
<tr>
<td>stop</td>
<td>14.0</td>
<td>15.4</td>
<td>23.1</td>
</tr>
<tr>
<td>Media latency</td>
<td>16.5</td>
<td>7.77</td>
<td>15.8</td>
</tr>
</tbody>
</table>

Table 7.2: Latency results for M-JPEG decoder component.

<table>
<thead>
<tr>
<th>Host</th>
<th>spitfire</th>
<th>tanglefoot</th>
<th>teteleys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>Mean (ms)</td>
<td>St. Dev.</td>
<td>Mean (ms)</td>
</tr>
<tr>
<td>powerOn</td>
<td>24.7</td>
<td>14.9</td>
<td>12.9</td>
</tr>
<tr>
<td>powerOff</td>
<td>19.3</td>
<td>21.7</td>
<td>14.2</td>
</tr>
<tr>
<td>start</td>
<td>13.4</td>
<td>12.1</td>
<td>16.1</td>
</tr>
<tr>
<td>stop</td>
<td>15.9</td>
<td>20.7</td>
<td>18.8</td>
</tr>
<tr>
<td>Media latency</td>
<td>38.6</td>
<td>386.9</td>
<td>7.87</td>
</tr>
</tbody>
</table>

Table 7.3: Latency results for multicast UDP sink component.
underlying OS. Many of these calls will block and schedule another thread on the CPU while the
operation completes, or spend an unpredictable amount of time executing OS code.

On the other hand, the media latency results show a clear performance improvement on the faster
host, especially in the case of the MPEG decoder. The media latency for the multicast UDP sink is
interesting, as it essentially reflects the sum of the network transit time plus any required unmarshalling
and de-fragmentation at the sink. The average latency for the two 450MHz and 500MHz
hosts is almost five times that of the 600MHz machine—more than can be accounted for solely by
the difference in raw CPU speed. These machines have identical Ethernet adapters to the 600MHz
host and connect directly to the same switch; thus it seems unlikely that the difference in latency
is a hardware issue. It may be that there is some subtle—yet important—variation in the software
configuration of the various machines, although I have so far been unable to verify this hypothesis.

7.2.3 Optimised trials

This section presents results of experimental trials using the optimisations described in Section 6.2.2.
The test application was run with two different sets of optimisations:

1. Fine-grained scheduling of the integration stage of a reconfiguration; specifically, scheduling
the injection of trigger events into the various reconfigured regions of the initial and final
path configurations.

2. Adjustment of the setup/integration stage boundary for individual reconfigured objects. The
current implementation is only able to move the stage boundary forward—that is, moving
reconfiguration actions from the integration stage to the setup stage—so this optimisation is
referred to as “early integration” below. In the trials shown here, the integration scheduling
optimisation was always active as well whenever the early integration optimisation was used.

“Change source” reconfiguration

Figure 7.17 shows the behaviour of the test application running the “change source” reconfiguration
with optimised scheduling. The layout and notation used on the graphs is identical to the non-
optimised trials above. Detailed views of the end-to-end latency over a single reconfiguration are
shown here; as with all of these experiments, complete results may be found in Appendix A.

The latency graphs for the different optimisation schemes are practically identical; they would also
be hard to distinguish from the non-optimised case shown in Figure 7.12(a). This experiment
suffers from the same problems discussed at length above: the high latency of the MPEG decoder
coupled with its buffering and jitter-control effectively mask most of the effects of this relatively
lightweight reconfiguration. As previously, the switch to the new stream appears to occur several
Figure 7.17: End-to-end latency for optimised "change source" reconfiguration with MPEG streams.
seconds after the reconfiguration completes, when in fact the data just prior to the indicated switch point is mis-labelled and in reality belongs to the second stream.

Sharp-eyed readers will notice that the reconfiguration using early integration apparently takes longer to complete than the scheduling optimisation-only trial (499ms vs. 309ms in this case). Both are slower than the non-optimised trial at 214ms. This is because the current optimised scheduler is somewhat conservative, typically waiting longer than necessary for the setup stage to complete before starting the integration stage. Of course, the length of a reconfiguration may be extended indefinitely if the enclosing atomic action is waiting for a particular event to trigger the integration stage. More interesting is the scheduling within the integration stage of the optimised reconfigurations. In both cases the final path configuration is integrated first, allowing data from the second stream to begin flowing along the path even while the initial path configuration is being shut down.

However, even considering only the integration stage of the two transactions, the early integration trial still appears to be slower, with a $t_{\text{integrate}}$ value of 383ms against 226ms for the scheduling-only trial. This is a frequent feature of the trials in this experiment and is at least partly because the network sink in the scheduling-only trial is able to take advantage of data buffered by its OS-level network socket (as discussed previously with respect to the non-optimised version of this experiment). Having several incoming media elements immediately available allows the integration trigger event to reach the downstream end of the path more rapidly than in the early integration case, where the sink is already running and must wait for fresh data to be delivered.

"Switch encodings" reconfiguration

Figures 7.18 and 7.19 show results for optimised trials of the "switch encodings" reconfiguration, for the M-JPEG→MPEG and MPEG→M-JPEG transitions respectively.

The optimisations have clearly had a beneficial effect in the case of the M-JPEG→MPEG transition: comparing the graphs in Figure 7.18 with their non-optimised equivalents in Figure 7.13 shows that the 2200ms pause in output shown in the latter has been almost entirely eliminated.

Performance in the trial utilising early integration is arguably the worse of the two, as there is a 160ms break in output before the first MPEG-encoded frame is seen, versus 42ms in the scheduling-only trial. However, considered in terms of the media latency of the MPEG decoder and, more importantly, the variation in that latency, both schemes have performed equally well. The real advantage of early integration is that it allows the reconfiguration to be completed quickly, at the expense of greater resource usage during the setup stage. If an application knows that a particular reconfiguration will be performed in the near future, use of early integration will in general lead to the visible transition between initial and final configuration being significantly faster, if not necessarily smoother. In this instance, the value of $t_{\text{integrate}}$ for the scheduling-only trial is 2651ms, while for the early integration trial it is 333ms.
Figure 7.18: End-to-end latency for optimised "switch encodings" reconfiguration for M-JPEG→MPEG transition.
Figure 7.19: End-to-end latency for optimised "switch encodings" reconfiguration for MPEG→M-JPEG transition.
Similar behaviour can be observed for the MPEG→M-JPEG transition in Figure 7.19. The graphs show a 32ms gap and 88ms overlap between the last MPEG and first M-JPEG frames for the scheduling-only and early integration trials respectively. The corresponding values of $t_{\text{integrate}}$ are 930ms and 506ms.

It should be noted that neither of the optimisation techniques used here offer a solution to the synchronisation issues encountered in the non-optimised trial. That is, if the last MPEG-encoded frame seen is frame $N$, there is no guarantee that the first M-JPEG frame will be frame $N + 1$. Any difference in latency along the two versions of the path will lead to a corresponding offset in the two versions of the stream data. A potential solution, involving the insertion of extra buffering components to equalise the latencies, was discussed in the previous chapter.

“Move client” reconfiguration

Finally, we come to the optimised trials of the “move client” reconfiguration, presented in Figures 7.20 and 7.21 for MPEG and M-JPEG source stream respectively.

In both MPEG trials, the scheduler has overestimated the time taken to start up the client on the second host by around 750ms, meaning that the new MPEG decoder will have started producing data before the old client has been shut down. The effect of this is more obvious in the scheduling-only optimisation case, since both clients will be running simultaneously for three-quarters of a second. In the second trial, with early integration active as well, the second decoder is started during the setup stage; any initial frames it produces are quietly dropped. The second display is not started until the integration stage runs, with the net result being a 181ms (around 2.5 frame times) pause in output, a glitch that is unlikely to be visible to an application user given that the second display is supposedly located some distance from the first.

The most likely explanation for the consistent overestimation of the MPEG decoder startup time observed here is that in both cases the source MPEG stream was on its first playback cycle; that is, it had yet to be rewound. At this point, the problems of input buffering and incorrect counting of frames that afflict the decoder component would not yet be having a great effect. Thus, the decoder initially exhibits a latency lower than the average used to compute the schedule. Note, however, that the measured latency rises soon after the reconfiguration, to a value closer to the recorded average.

Turning to the M-JPEG trials for this reconfiguration, Figure 7.21(a) shows an almost perfect reconfiguration, with the switch occurring in 81ms—slightly more than a single frame time. The only anomaly is the relatively long time taken to complete integration of the initial path configuration; however, most of this activity will not be visible to the user. Figure 7.21(b) again shows the effect of any delay before the integration stage is triggered, when early integration is used: the M-JPEG...

---

*Unfortunately, in this particular instance, buffering effects led to a minor loss of synchronisation between the two hosts, such that three frames were dropped during the reconfiguration and not seen on either display.*
Figure 7.20: End-to-end latency for optimised "move client" reconfiguration with MPEG stream.
Figure 7.21: End-to-end latency for optimised "move client" reconfiguration with M-JPEG stream.
decoder on the second host starts to generate output frames, although these are dropped without ever being displayed.

7.3 Summary

This chapter has presented a preliminary experimental evaluation and validation of the architecture presented in previous chapters. A test application, derived from the remote surveillance case study described in Chapter 3 was implemented and used to demonstrate a selection of typical reconfiguration scenarios. These scenarios were both run with and without the reconfiguration scheduling optimisations introduced in Chapter 6. Detailed traces of the test application’s behaviour under each of these reconfigurations were recorded and analysed for presentation in this chapter.

The results of the non-optimised trials demonstrate that the architecture, as implemented in the DJINN framework, does in fact work: an application model is able to reliably and repeatably carry out reconfigurations on running, remote components. As expected, the performance of the non-optimised reconfigurations—in smoothness terms—is often considerably less than perfect. The blame for a certain amount of this performance gap can be laid on the underlying non-real time platform as well as known deficiencies in particular active-layer components; however, the bulk of the responsibility must be attached to the naïve basic scheduling algorithm.

The optimised trials of Section 7.2.3 show that quite significant performance gains can be realised by the application of relatively simple enhancements to the scheduling algorithm. The greatest improvements were achieved by taking account of the latency inherent in each active layer action and scheduling the final stage of each reconfiguration appropriately. Use of the "early integration" optimisation often reduced the smoothness of reconfigurations relative to the scheduling optimisation alone; however, it should be re-stated that the purpose of early integration is to hide the latency of time-consuming reconfigurations—typically with additional resource cost—rather than to explicitly improve smoothness.

While this chapter has highlighted a number of issues with both the reconfiguration architecture and the DJINN implementation, it has not proposed any specific areas for further investigation. This topic will be covered in the next chapter.
CHAPTER 8

SUMMARY AND CONCLUSIONS

The overall motivation for this research is summarised by the following passage from Chapter 1:

"... multimedia applications present a unique set of requirements to distributed systems architects, primarily due to the real-time processing constraints of continuous media streams and the dynamic reconfiguration and multiuser interaction that characterises the application domain. Constructing mission-critical systems that meet these requirements poses significant difficulties. I believe that this problem is best attacked through the development of a comprehensive support framework for distributed multimedia systems that provides generic solutions to the shared needs of these applications. The challenge laid before distributed systems and RTOS researchers is to develop generic, highly programmable platforms supporting the dual goals of predictable real-time behaviour alongside flexible, interactive multimedia processing."

The preceding five chapters have presented an experimental architecture that attempts to address this problem, with particular emphasis on the dynamic reconfiguration of active, distributed multimedia systems. This final chapter presents a summary of the thesis, with reference to the contributions claimed in the introduction as well as a number of unresolved issues, and discuss various directions for future work in this area.

8.1 Summary

This thesis has presented a novel architecture supporting the construction, execution and dynamic reconfiguration of distributed, component-based multimedia applications. The architecture utilises a two-layer application model to provide a separation of concerns between application structure and component implementation. Reconfigurations are encapsulated within transaction-like atomic
actions performed against the application model. Updates to the active components of an application are scheduled so as to maximise the smoothness of the transition between the initial and final configurations.

8.1.1 Contributions

Application modelling

Chapter 3 developed a comprehensive set of characteristics and requirements for the class of distributed multimedia applications addressed by this research. The requirements were largely derived from analysis of two detailed application scenarios: an existing remote surveillance system described in the literature and a hypothetical but feasible system for a digital television production environment.

Following directly from the analysis of requirements, Chapter 4 presented an architecture for composing, controlling and reconfiguring distributed multimedia applications. This was discussed not only in terms of its high-level abstractions, but also in the context of the prototype DJINN programming framework that implements many aspects of the architecture. The novelty of the DJINN architecture lies in its separation of application components into two distinct layers. The upper model layer forms a dynamic, runtime model of the underlying media-processing components in the active layer, encapsulating application structure, QoS control and reconfiguration functionality. Application users and programmers work exclusively with the model; the framework itself manages all interactions with the active components.

I believe that this architecture meets my goal of providing a flexible, generic platform for dynamically reconfigurable distributed multimedia applications.

Reconfiguration techniques

The application modelling architecture was extended in Chapters 5 and 6 to support reliable, consistent reconfiguration of running applications. Reconfigurations are encapsulated within atomic actions—transactional constructs that ensure the application moves atomically from one consistent configuration to another, or remains in its initial configuration if the transition cannot be performed successfully. Validation of atomic actions takes place entirely within the application model; the active layer is not modified until it is known that the reconfiguration can be completed safely.

The final phase of an atomic action—mapping changes to the application model onto the corresponding active layer objects—must be done carefully to avoid breaking the dynamic integrity of the application, in the form of lost or corrupted data or observable glitches in output. This is achieved by scheduling updates to the active layer in such a way that the existing configuration is not stopped or otherwise disrupted until the transition to the final configuration is ready to take place. The
schedule expresses a tradeoff between the perceived smoothness of the transition, the time taken to complete the reconfiguration, and the additional resources consumed during the reconfiguration process.

Together these mechanisms provide a robust solution to many of the problems inherent in dynamically reconfiguring active continuous-media applications.

Validation

Chapter 7 presented an experimental evaluation of the architectural abstractions developed in the preceding chapters, as well as their implementation in the prototype DJINN framework. The evaluation was performed using a simple test platform derived from the remote surveillance application scenario introduced as part of the requirements analysis in Chapter 3. The test application allowed a variety of typical structural and non-structural reconfigurations to be executed on a collection of active media streams; detailed traces of individual active object behaviour were gathered over multiple trials in a range of different application scenarios. Experiments were run using both the basic, non-optimised reconfiguration algorithm, and with integration stage scheduling and early integration optimisations active.

Overall, the system performed surprisingly well, especially considering that it is written largely in Java and was running on a non-real time OS and network platform with minimal quality of service support. The non-optimised trials demonstrate that the modelling and reconfiguration architecture implemented by DJINN is fundamentally sound: distributed applications can be successfully constructed and reconfigured through the application model and atomic action mechanism. The reconfigurations themselves generally behaved as predicted, although a number of serious flaws in the implementation of certain active components were discovered.

Results from the optimised trials validate the hypothesis that significant performance gains can be achieved through relatively straightforward analysis and manipulation of reconfiguration actions. Scheduling the integration stage of the reconfigurations succeeded in eliminating many of the temporal integrity violations observed in the non-optimised experiments. The early integration technique was also able to consistently reduce the apparent latency of reconfigurations, at the expense of increased resource load during the setup stage.

8.1.2 Unresolved issues

While the contributions listed above have successfully addressed the primary research goals of this thesis, there remain a number of areas where the existing DJINN architecture and framework are inadequate. Some of these issues were noted in Chapter 3 as requirements that fell outside the scope of this research; others have become apparent during testing and validation of the architecture.
Application modelling

Modelling complex applications and components may turn out to be more difficult than I had at first supposed; if this is the case, the existing framework abstractions could prove to be inadequate for the task. The potential difficulty can be seen even in the simple test application used to validate the framework. Consider, for example, the MPEG codec components. The behaviour of these components is highly data-dependent; that is, dependent on the actual content of the data stream fed to the component as well as more coarse-grained characteristics such as frame size and rate. Analysing the stream contents in order to accurately predict component behaviour may not be possible—the stream may be coming from a live source, for example. Furthermore, even when such analysis is possible, it will likely be too time-consuming to be useful in many situations.

Applications may also exhibit interactions or dependencies between components that are awkward to model under the current architecture, especially where such dependencies cross composite component boundaries. DJINN attempts to adhere to a rule that components do not know or care what they are connected to, because their behaviour will be the same in any case. Clearly this is unlikely to be strictly true in every situation; however, the existing modelling abstractions cannot express all of the potential interactions between components.

Genericity

One of the stated requirements of the architecture is that it be generic—able to support any application within the target domain, including those not considered by its designers. The existing architecture does not fully meet this requirement. In the first place, applications remain largely tied to the particular environment they were written for; that is, a specific collection of hosts, networks and multimedia devices. Admittedly it is trivial to modify an application to operate in a new environment—it need not even involve changing the Java source code—but this still requires manual intervention, whereas the original vision was of truly generic applications that could be instantiated anywhere and adapt automatically to their environment.

Secondly, as the previous points concerning modelling illustrate, not all target applications can be fully modelled by the current framework. An application that cannot be accurately modelled may not run correctly, or may operate at a degraded level of service.

I suspect that these deficiencies in the modelling abstractions stem from the framework having been designed and built in a somewhat ad-hoc fashion. The design was largely driven by application requirements; specifically application scenarios such as those in Chapter 3 that were deemed to be somehow representative of the target application domain. What has now become clear is that these scenarios have not sufficiently illuminated all of the issues pertinent to modelling the target application class. It seems likely that the architecture would benefit from a more formal design
process, working from abstract specifications of application characteristics and requirements rather than concrete examples.

Validation

Although the test application used in Chapter 7 is based on an existing application, it cannot make a particularly strong claim to represent "real world" conditions. Thus, there is a need for further evaluation of the architecture with a larger and more complex application, ideally one with real users.\footnote{DJINN has been used to build a video-conferencing system with “follow-me” capabilities supported by Active Badge technology [HH94]. It also forms part of the platform for the QoS DREAM project at Cambridge University [MSBC98] which is developing an integrated communications system for clinicians in an emergency ward.}

General distributed systems issues

The scalability performance of the DJINN architecture with respect to larger applications and greater numbers of components is still in question. Although Section 3.3.1 (Page 60) explicitly excluded scalability concerns from the scope of this work, there is no doubt that this will become an issue if the DJINN abstractions are to be used in real world applications. Use of federated models (Section 4.2.4, Page 77) will go some way towards reducing the apparent complexity of an application model, at least from the point of view of each federation member. Beyond this, however, the ability of the framework abstractions—in particular the reconfiguration and QoS management subsystems—to cope with orders of magnitude increases in component numbers is unknown.

Similarly, the architecture at present makes no attempts to provide any form of fault-tolerance or orderly recovery from failures. I would argue that failure management is a general problem for all distributed systems, and that there are many existing solutions that might be suitable for DJINN—obvious possibilities include replicated models, persistent models and transactional mechanisms to provide atomic recovery of model and active layer state after a failure. Nevertheless, as with scalability, fault-tolerance will become an important consideration if the framework is to honestly claim support for "mission critical" applications.

8.2 Directions for future work

In addition to the major issues outlined above, evaluation of the DJINN architecture has revealed other, less critical deficiencies in the current design and implementation. Furthermore, a number of promising directions for future research can be identified, to further explore and extend the work presented here:
Modelling

There are numerous ways in which the application modelling concept could be extended. With respect to the need identified above for a more formal design analysis, it would be beneficial to decouple the specification of a model's structure, QoS and reconfiguration properties from the underlying Java implementation. A high-level model representation could take the form of a specialised language or notation, or make use of a graphical tool for designing and manipulating models. The model would also benefit from improved support for multiple paths within a single application, including the ability to specify synchronisation and other relationships between paths.

Resource database

A significant obstacle to the creation of truly generic application models is a lack of information about the execution environment, particularly the available multimedia devices and CPU/network resources. One approach to this problem is to provide a resource database that generic model objects can query to determine a suitable configuration for their active counterparts. Such a database would clearly store details of available hardware and software resources, but might also integrate data from other sources, for instance users' locations from tracking devices such as Active Badges or GPS receivers.

Reconfiguration

The existing reconfiguration subsystem requires numerous improvements, especially with regard to the active layer update process. An important area of ongoing research is the extension of the scheduling algorithms to support more complex, multiple-path reconfigurations, as well as finer-grained optimisation of the schedules themselves. Alongside this work, there is a need for a more comprehensive analysis of the smoothness/resource/time tradeoff, which can be fed back into the design of the scheduling algorithms. Finally, there is a more distant possibility of implementing long-lived optimistic reconfigurations of the type described in Section 5.2.

QoS integration

Improved quality of service support is necessary at two levels: firstly, accurate modelling of component's QoS characteristics—although, as discussed above, this is a significant challenge in itself. Secondly, implementation of the active layer over a suitable RTOS, ideally one supporting management of network as well as host resources.
Other implementation issues

The analysis in Chapter 7 indicated a number of problems with the existing active layer implementation. Clearly certain active component classes are in need of replacement. More fundamentally, it may be advantageous from a purely performance-oriented viewpoint to eliminate Java from the active layer entirely. This is not as drastic a step as it may appear, as it would merely continue the trend, started in the current prototype, of migrating media processing functions to native C or C++ code. It would also be necessary to replace the RMI-based model→active interface with one using CORBA or, to minimise disruption to the existing model implementation, RMI over IIOP.

8.3 Concluding remarks

The goal of this research has been to investigate and develop architectural support for distributed multimedia systems, providing a generic platform for mission-critical applications incorporating continuous media streams and dynamic online reconfiguration. Underlying this is the hypothesis that a programming framework and middleware layer are appropriate vehicles to meet the needs of these applications, and more specifically that an approach based on the use of a dynamic runtime model of the application is both feasible and practical.

I believe that the results of the investigation show this hypothesis to be essentially valid: the architecture as designed and implemented does realise its fundamental goals, in that distributed continuous media applications can be constructed at a relatively high level of abstraction, then instantiated and reconfigured with substantially less programmer effort than a custom solution implemented without the benefit of middleware support.

Clearly, the architecture presented here is far from being a complete panacea for all the problems encountered by developers of distributed multimedia applications; there is a considerable amount of work to be done before the platform can be deemed truly generic and able to cope with the rigours of real world applications. However, I submit that this work does represent a step forward and trust that it will prove useful as a basis for future experimentation and research.
APPENDIX A

EXPERIMENTAL RESULTS

This appendix presents selected results from my experimental validation of the DJINN framework prototype, in addition to those found in Chapter 7. For reasons of space it is impractical to include the results of every trial; thus, I have chosen a representative trial from each experiment for inclusion here. The results for the other trials were broadly similar within each experiment.

A.1 Non-optimised trials

A.1.1 Baseline operation

Figures A.1–A.6 show the behaviour of the test application in continuous operation, playing MPEG and M-JPEG encoded versions of the same video stream, respectively. The figures show the media rate (equal to the frame rate in these trials), end-to-end latency and end-to-end jitter along the video source–display path, over the entire duration of the trial as well as more detailed views over a 30-second window. The interval used in the 30-second views was chosen to show an instance of the video stream “rewinding” before looping back to the first frame again.

The media rate is calculated with respect to a one second time window leading up to the receipt of each media element. Likewise, the end-to-end latency shown in the figures is a one second rolling average using the same sliding window. The jitter is calculated relative to this rolling average.
Figure A.1: Media rate for baseline operation with MPEG stream.
Figure A.2: End-to-end latency for baseline operation with MPEG stream.
Figure A.3: End-to-end jitter for baseline operation with MPEG stream.
Figure A.4: Media rate for baseline operation with M-JPEG stream.
Figure A.5: End-to-end latency for baseline operation with M-JPEG stream.
Figure A.6: End-to-end jitter for baseline operation with M-JPEG stream.
A.1.2 Non-structural reconfiguration

Figures A.7–A.12 are similar to the corresponding figures from the previous experiment, but show the test application undergoing non-structural reconfigurations (stopping and restarting the client components). In these figures the "zoomed" views show a single pair of reconfigurations in greater detail; the times at which the setup and integration stages of each reconfiguration were initiated by the application model and completed by the active layer are shown on these views.
(a) Complete trial.

(b) Single reconfiguration.

Figure A.7: Media rate for non-structural reconfiguration with MPEG stream.
Figure A.8: End-to-end latency for non-structural reconfiguration with MPEG stream.
Figure A.9: End-to-end jitter for non-structural reconfiguration with MPEG stream.
Figure A.10: Media rate for non-structural reconfiguration with M-JPEG stream.
Figure A.11: End-to-end latency for non-structural reconfiguration with M-JPEG stream.
Figure A.12: End-to-end jitter for non-structural reconfiguration with M-JPEG stream.
A.1.3 "Change source" reconfiguration

Results for the first of the structural reconfigurations (switching source streams with a single client) are shown in Figures A.13–A.15, for MPEG encoded streams only. As with the non-structural reconfiguration the figures show media rate, latency and jitter data for the complete trial as well as detail views of a single reconfiguration.
Figure A.13: Media rate for "change source" reconfiguration with MPEG streams.
(a) Complete trial.

(b) Single reconfiguration.

Figure A.14: End-to-end latency for "change source" reconfiguration with MPEG streams.
Figure A.15: End-to-end jitter for "change source" reconfiguration with MPEG streams.
A.1.4 "Switch encodings" reconfiguration

The "switch encodings" reconfiguration is presented in Figures A.16–A.21. The first set of three figures show a complete trial including both M-JPEG→MPEG and MPEG→M-JPEG reconfigurations. Each of the transition types is shown in greater detail by the remaining graphs.
Figure A.17: End-to-end latency for “switch encodings” reconfiguration, complete trial.

Figure A.18: End-to-end jitter for “switch encodings” reconfiguration, complete trial.
Figure A.19: Media rate for "switch encodings" reconfiguration, detail.
Figure A.20: End-to-end latency for "switch encodings" reconfiguration, detail.
Figure A.21: End-to-end jitter for "switch encodings" reconfiguration, detail.
A.1.5 "Move client" reconfiguration

Finally, Figures A.22–A.27 illustrate the "move client" reconfiguration for both MPEG and M-JPEG streams. The layout of the graphs is identical to that of the previous experiments.
Figure A.22: Media rate for "move client" reconfiguration with MPEG stream.
Figure A.23: End-to-end latency for "move client" reconfiguration with MPEG stream.
Figure A.24: End-to-end jitter for “move client” reconfiguration with MPEG stream.
Figure A.25: Media rate for "move client" reconfiguration with M-JPEG stream.
Figure A.26: End-to-end latency for "move client" reconfiguration with M-JPEG stream.
Figure A.27: End-to-end jitter for "move client" reconfiguration with M-JPEG stream.
A.2 Optimised trials

A.2.1 "Change source" reconfiguration

Figures A.28–A.30 show the behaviour of the "change source" reconfiguration with optimised integration scheduling active (the algorithm used for scheduling the integration of the initial and final path configurations is described in Section 6.2.2). Likewise, Figures A.31–A.33 illustrate the same reconfiguration program, in this case using the "early-integrate" optimisation—that is, adjusting the setup-integration stage boundary for individual path components to reduce the latency of the integration stage—in addition to the scheduling optimisation. The layout of the graphs is identical to that used for the non-optimised trials above.
Figure A.28: Media rate for “change source” reconfiguration with MPEG streams, scheduling optimisation only.
Figure A.29: End-to-end latency for "change source" reconfiguration with MPEG streams, scheduling optimisation only.
Figure A.30: End-to-end jitter for "change source" reconfiguration with MPEG streams, scheduling optimisation only.
Figure A.31: Media rate for "change source" reconfiguration with MPEG streams, scheduling and early-integrate optimisations.
Figure A.32: End-to-end latency for "change source" reconfiguration with MPEG streams, scheduling and early-integrate optimisations.
Figure A.33: End-to-end jitter for “change source” reconfiguration with MPEG streams, scheduling and early-integrate optimisations.
Figure A.34: Media rate for "switch encodings" reconfiguration with scheduling optimisation, complete trial.

A.2.2 "Switch encodings" reconfiguration

Figures A.34–A.39 and A.40–A.45 show sample results for the "switch encodings" reconfiguration, with scheduling optimisation only, and scheduling optimisation plus early integration, respectively.
Figure A.35: End-to-end latency for "switch encodings" reconfiguration with scheduling optimisation, complete trial.

Figure A.36: End-to-end jitter for "switch encodings" reconfiguration with scheduling optimisation, complete trial.
Figure A.37: Media rate for “switch encodings” reconfiguration with scheduling optimisation, detail.
Figure A.38: End-to-end latency for "switch encodings" reconfiguration with scheduling optimisation, detail.
Figure A.39: End-to-end jitter for "switch encodings" reconfiguration with scheduling optimisation, detail.
Figure A.40: Media rate for “switch encodings” reconfiguration with scheduling and early-integrate optimisations, complete trial.
Figure A.41: End-to-end latency for "switch encodings" reconfiguration with scheduling and early-integrate optimisations, complete trial.

Figure A.42: End-to-end jitter for "switch encodings" reconfiguration with scheduling and early-integrate optimisations, complete trial.
Figure A.43: Media rate for "switch encodings" reconfiguration with scheduling and early-integrate optimisations, detail.
Figure A.44: End-to-end latency for "switch encodings" reconfiguration with scheduling and early-integrate optimisations, detail.
Figure A.45: End-to-end jitter for “switch encodings” reconfiguration with scheduling and early-integrate optimisations, detail.
A.2.3 "Move client" reconfiguration

Results for the "move client" reconfiguration with optimisations are given in Figures A.46–A.51 and A.52–A.57, for both MPEG and M-JPEG source streams. The first set of graphs were obtained by running the test application with only the integration scheduling optimisation active, while the second set utilised both scheduling and early integration optimisations.
Figure A.46: Media rate for "move client" reconfiguration with MPEG stream, scheduling optimisation only.
Figure A.47: End-to-end latency for “move client” reconfiguration with MPEG stream, scheduling reconfiguration only.
Figure A.48: End-to-end jitter for "move client" reconfiguration with MPEG stream, scheduling reconfiguration only.
Figure A.49: Media rate for "move client" reconfiguration with M-JPEG stream, scheduling optimism only.
Figure A.50: End-to-end latency for "move client" reconfiguration with M-JPEG stream, scheduling optimisation only.
Figure A.51: End-to-end jitter for "move client" reconfiguration with M-JPEG stream, scheduling optimisation only.
Figure A.52: Media rate for "move client" reconfiguration with MPEG stream, scheduling and early-integrate optimisations.
Figure A.53: End-to-end latency for “move client” reconfiguration with MPEG stream, scheduling and early-integrate optimisations.
Figure A.54: End-to-end jitter for "move client" reconfiguration with MPEG stream, scheduling and early-integrate optimisations.
Figure A.55: Media rate for "move client" reconfiguration with M-JPEG stream, scheduling and early-integrate optimisations.
Figure A.56: End-to-end latency for “move client” reconfiguration with M-JPEG stream, scheduling and early-integrate optimisations.
Figure A.57: End-to-end jitter for “move client” reconfiguration with M-JPEG stream, scheduling and early-integrate optimisations.
BIBLIOGRAPHY


[BBRVD98] Luc Bolliard, Fabienne Boyer, Michel Riveill, and Jean-Yves Vion-Dury. System services for distributed application configuration. In Proc. ICCDS '98 [Pro98b].


IBM European Networking Center, ACM SIGCOMM, ACM SIGOPS, Springer-Verlag, Berlin, Germany.


[LMB+97] Ian Leslie, Derek McAuley, Richard Blank, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7), September 1997.


