

# **Streaming Geospatial Imagery into Virtual Environments**

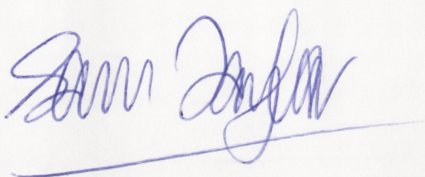
**Samuel Roger Aldiss Taylor**

A thesis submitted for the degree of  
Doctor of Philosophy at  
The Australian National University

August 2001

Except where otherwise indicated, this thesis is my own original work.

Samuel Roger Aldiss Taylor  
6 August 2001

A handwritten signature in blue ink, appearing to read "Sam Taylor", is written above a single horizontal blue line.

**For Mum, Dad and Lucy.**

---

# Acknowledgements

---

A PhD is a long road to tread, and I owe a great debt of thanks to all those who have helped me over the last four years. I have been privileged to study at the Australian National University, an institution which continues to inspire generations of young Australians. Within the Department of Computer Science I must thank my supervisory panel and in particular Dr Chris Johnson, who always made time for me and always challenged me to think in new directions.

Special mention has to go to the Cooperative Research Centre for Advanced Computational Systems (ACSys). ACSys was a remarkable organisation: it gave me the opportunity to work with great people on fascinating problems using state-of-the-art technology, and ultimately to develop this dissertation. Beyond that I made some wonderful friends through ACSys: John Zigman, Matthew Wilson, Markus Buchhorn, Linda Wallace, Raj Nagappan, Nick Craswell, Dave Walsh, Hugh Fisher, Zhen He, Luke Kirby, Alonso Marquez, Steve Blackburn and Jan Bitmead.

Away from work there are just as many kind people who helped and encouraged me. I have been lucky enough to be part of two wonderful caring families, the Taylors and the Jensens, both of whom offered their unconditional love and support. My mates helped me maintain my sense of perspective, each in their own unique way. Angus O'Shea introduced me to the world of Belgian beers, John Zigman got me riding and forced me in to the gym, Tim Edwards showed me just how much you can achieve when you really want to, and Gavin Longhurst fed me a steady diet of mad ideas and exquisite language. Then there were the lunchtime coffee drinkers – Gavin Mercer, Andrew Slater and Matt Taylor – who prove that nothing beats a good conversation and a strong espresso sat outside in the Canberra sunshine.

Finally Richard Walker's refined aesthetic sense, remarkable knowledge of  $\LaTeX$  and very great patience made it possible for me to typeset this thesis.

---

# Abstract

---

The goal of this thesis is to enable geospatial images from earth observation satellites to be visualised in a Collaborative Virtual Environment (CVE). Geospatial images are a valuable commodity and are used in fields as diverse as meteorology, defence, environmental impact analysis and urban planning. They are also very large, and collections of images are typically kept on slow tertiary storage devices. Collaborative Virtual Environments are sophisticated visualisation systems with the added attraction of human interaction. They are also computationally demanding applications which must respond to user input in real time. So, while there is great value in combining CVEs and geospatial imagery, a range of performance and management problems must first be considered.

The general challenge is to retrieve imagery from storage archives, transform it through various image processing operations, and disseminate it to visualisation clients with a minimum of latency. This challenge implies four fundamental requirements: for *responsive* client access to imagery, for high rates of *throughput* when disseminating imagery, for *sharing collaborative data* between clients, and for a structured approach to *application management*.

The major contribution of this thesis is to report a software architecture that addresses each of these requirements. Known as the Responsive Architecture for Pipelined Imagery Dissemination (RAPID), it consists of three main components:

1. A general-purpose dissemination pipeline is used to access and process imagery at high rates of throughput.
2. Large, parallel caches are deployed close to visualisation clients to decouple them from the pipeline and provide responsive access to processed imagery. These caches are also organised to route collaborative data between sites.
3. An application management framework is defined to allow dissemination pipelines to be created within a computational grid.

The RAPID architecture has been validated through the development of numerous working systems. Three case studies are presented, which allow general architectural principles to be distilled from practical work.

The sheer size of geospatial imagery will continue to present a number of difficult problems for developers of visualisations systems: the underlying tension between client responsiveness and archive latency is not easily resolved. However, the results of this thesis demonstrate that the problems are not insurmountable, and that real-time collaborative visualisation of geospatial imagery is an achievable goal.

---

# Contents

---

<b>Acknowledgements</b>	<b>vii</b>
<b>Abstract</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Data Domain: Geospatial Imagery . . . . .	1
1.2 Visualisation Domain: Collaborative VE . . . . .	2
1.3 The Problem Framed as Four Functional Requirements . . . . .	5
1.3.1 Responsiveness . . . . .	5
1.3.2 Throughput . . . . .	5
1.3.3 Collaborative Data Sharing . . . . .	6
1.3.4 Application Management . . . . .	6
1.4 Scope and Suitability of the Problem . . . . .	7
1.5 A Solution . . . . .	8
1.6 Approach and Methodology . . . . .	8
1.7 Organisation . . . . .	9
<b>2 Formative Concepts</b>	<b>11</b>
2.1 Managing and Using Geospatial Imagery . . . . .	11
2.1.1 Earth Observation Systems and Data . . . . .	11
2.1.2 Storage and Repositories . . . . .	12
2.1.3 Image Processing and Dissemination Pipelines . . . . .	13
2.1.4 Tiling and Resolution . . . . .	15
2.1.5 Map Projections and Coordinate Systems . . . . .	17
2.2 Challenges with Collaborative Virtual Environments . . . . .	18
2.2.1 Data Distribution Tradeoffs . . . . .	18
2.2.1.1 Model/View Distinction . . . . .	19
2.2.1.2 Data Size . . . . .	20
2.2.1.3 Access and Communications Patterns . . . . .	20
2.2.1.4 Computational Cost . . . . .	21
2.2.1.5 Time Sensitivity and Buffering . . . . .	21
2.2.1.6 Scalability . . . . .	22
2.2.2 Object Position Estimation . . . . .	22
2.2.3 Perceptual Consistency and Collaborative Data Sharing . . . . .	23
2.2.4 Terrain Rendering . . . . .	24
2.3 Distributed Computing . . . . .	27
2.3.1 Data Streaming . . . . .	27

---

2.3.2	Caching . . . . .	28
2.3.3	Application Management, Metacomputing and Grids . . . . .	29
2.4	Summary . . . . .	31
<b>3</b>	<b>Techniques and Approach</b>	<b>33</b>
3.1	Runtime Techniques . . . . .	33
3.1.1	Pipelined Imagery Dissemination . . . . .	33
3.1.2	Pipeline Scheduling . . . . .	38
3.1.3	Parallel Caches at Visualisation Sites . . . . .	40
3.1.4	Group Communications for Collaborative Data Sharing . . . . .	42
3.1.5	Inter-site Tile Sharing . . . . .	43
3.1.6	Parallel Streaming in the Dissemination Pipeline . . . . .	43
3.1.7	Approximating Tiles . . . . .	47
3.2	Grid-based Application Management . . . . .	48
3.3	Bringing the Pieces Together . . . . .	52
3.3.1	Illustrative Example . . . . .	53
3.3.2	Negotiating a Service . . . . .	53
3.3.3	Initialisation of Services . . . . .	56
3.3.4	Provision of Services . . . . .	57
3.3.5	Disposal of Services . . . . .	59
3.4	Caveats . . . . .	59
3.5	Summary . . . . .	60
<b>4</b>	<b>RAPID: Responsive Architecture for Pipelined Imagery Dissemination</b>	<b>63</b>
4.1	Documenting An Architecture . . . . .	63
4.1.1	UML and Perspectives . . . . .	65
4.1.2	Implementation Profiles . . . . .	65
4.1.3	The UML Object Model and New Stereotypes . . . . .	66
4.2	Service Negotiation and Application Management . . . . .	67
4.2.1	Actors and the Pipeline Life-cycle . . . . .	68
4.2.2	Negotiating Access to a Resource . . . . .	70
4.2.3	Managing Access to Special Purpose Networks . . . . .	73
4.2.4	A Schema for Resource Discovery . . . . .	73
4.3	Creating the Pipeline During Service Initialisation . . . . .	76
4.3.1	The Structure of a New Operator . . . . .	77
4.3.2	Connecting Ports and Creating Links . . . . .	79
4.4	Provision of Service with a Dissemination Pipeline . . . . .	81
4.4.1	Basic Network Communications . . . . .	81
4.4.2	RAPPORT: The RAPID Operator Protocol . . . . .	84
4.4.3	Port Buffers and Approximating Results . . . . .	87
4.4.4	Handling and Processing Tile Requests . . . . .	88
4.4.5	Supporting Parallelism in the Pipeline . . . . .	91
4.4.5.1	Intra-operator parallelism . . . . .	92
4.4.5.2	Parallel Streaming Through Downstream Gathering . . . . .	92

---

4.4.5.3	Parallel Streaming Through Upstream Scattering . . .	94
4.5	RACE: The RAPID Cache . . . . .	95
4.5.1	Structure of the RACE . . . . .	95
4.5.2	Nodes and Ports in a RACE . . . . .	98
4.5.3	Handling Requests . . . . .	99
4.5.4	Scheduling Speculative Requests . . . . .	100
4.5.5	Inter-site Bus and Collaborative Data Sharing . . . . .	101
4.6	Minor Details and Points of Clarification . . . . .	102
4.6.1	How Clients Connect to a Pipeline . . . . .	102
4.6.2	Termination of Service . . . . .	102
4.6.3	Tiling and Level of Detail Mechanisms . . . . .	103
4.6.4	Observability . . . . .	103
4.7	Summary . . . . .	103
<b>5</b>	<b>Comet: a Study in Client Responsiveness</b>	<b>105</b>
5.1	Overview of Comet . . . . .	106
5.2	Requirements of Comet . . . . .	106
5.3	Architecture of Comet . . . . .	108
5.3.1	Data Distribution and the Server Cluster . . . . .	108
5.3.2	The Rendering Process . . . . .	110
5.3.3	Client Caching Strategies . . . . .	111
5.3.4	Multithreading and Quasi-asynchronous Data Delivery . . . . .	111
5.4	Analysis of Comet . . . . .	112
5.4.1	Test Environment and Metrics . . . . .	112
5.4.2	Caching . . . . .	113
5.4.3	Tile Sizes for Rendering and Distribution . . . . .	115
5.4.4	Multithreading and Asynchronous Communication . . . . .	117
5.4.5	Level of Detail and Cache Utilisation . . . . .	119
5.4.6	Network Bandwidth Usage . . . . .	119
5.5	Summary . . . . .	121
<b>6</b>	<b>Short Studies in Dissemination and Throughput</b>	<b>123</b>
6.1	OATS: Study of an Imagery Archive Browser . . . . .	124
6.1.1	Overview and Requirements . . . . .	124
6.1.2	Architecture and Implementation . . . . .	126
6.1.3	Analysis . . . . .	128
6.1.4	Summary of OATS . . . . .	128
6.2	IMAD: Pipelined Imagery Dissemination . . . . .	128
6.2.1	Overview and Requirements . . . . .	129
6.2.2	Experimental Analysis . . . . .	130
6.2.2.1	Communications Throughput . . . . .	131
6.2.2.2	Efficiency of Loop-back Communications . . . . .	133
6.2.3	Architecture and Relationship to RAPID . . . . .	133
6.2.4	Analysis of the Architecture . . . . .	136



---

6.2.5	Subsequent Work . . . . .	138
6.2.6	Summary of IMAD . . . . .	139
6.3	CROP: Optimising Throughput . . . . .	139
6.3.1	Overview and Requirements . . . . .	140
6.3.2	Architecture . . . . .	141
6.3.3	Analysis . . . . .	142
6.3.4	Summary of CROP . . . . .	143
6.4	Summary . . . . .	143
<b>7</b>	<b>vGrid: a Study in Application Management</b>	<b>145</b>
7.1	Overview of the vGrid . . . . .	145
7.1.1	Motivation . . . . .	145
7.1.2	Requirements of Collaborative Virtual Environments . . . . .	147
7.1.3	Goals . . . . .	147
7.2	Relationship to RAPID . . . . .	148
7.3	Architecture . . . . .	148
7.3.1	Resource Schema . . . . .	149
7.3.2	Factories and Traders . . . . .	151
7.3.3	Implementation . . . . .	153
7.4	Analysis of vGrid . . . . .	155
7.4.1	Success of vGrid . . . . .	155
7.4.2	Virtual World Management . . . . .	155
7.4.3	Static and dynamic configuration . . . . .	156
7.4.4	Schema Encoding . . . . .	157
7.4.5	Interoperability . . . . .	157
7.5	Summary . . . . .	158
<b>8</b>	<b>Conclusions and Future Work</b>	<b>161</b>
8.1	Summary . . . . .	161
8.2	Future Directions for Research . . . . .	162
8.2.1	A Full Implementation of RAPID . . . . .	162
8.2.2	Evaluating Scheduling and Prefetch Policies . . . . .	163
8.2.3	High Level User Interaction and Collaboration . . . . .	163
8.2.4	Other Issues . . . . .	163
8.3	Conclusions . . . . .	164
	<b>Appendices</b>	<b>165</b>
<b>A</b>	<b>RAPID Class Reference</b>	<b>167</b>
A.1	Service Negotiation Classes . . . . .	167
A.2	Operator and Pipeline Classes . . . . .	171
A.3	RAPPORT Classes . . . . .	178
A.4	RACE Classes . . . . .	183

---

<b>B UML Notation and Conventions</b>	<b>189</b>
B.1 Class Diagrams . . . . .	189
B.2 State Diagrams . . . . .	190
B.3 Sequence Diagrams . . . . .	190
<b>Bibliography</b>	<b>191</b>
<b>Index</b>	<b>205</b>



---

# Introduction

---

This thesis is about using Collaborative Virtual Environments (CVEs) to explore imagery from earth observation satellites. CVEs are extremely sophisticated visualisation systems. They completely immerse the user in a virtual space where the peripheral vision is saturated and the perception of depth is accurate. Not only are they compelling display devices, they have the added dimension of human interaction and so are a powerful forum for teams of experts to visualise and solve problems. Geospatial imagery is among the richest and most valuable visual data, and often requires careful analysis by experts in multiple domains. So there is great potential to use Virtual Environments for collaborative visualisation of geospatial imagery.

In a perfect world we would be able to access and interact with geospatial imagery as easily as we access a remote website or interact with a printer on a local area network. Before this potential can be realised, however, there are significant performance and resource management problems which must be solved. Geospatial images are large and difficult to manage and distribute. They also require considerable processing before they can be visualised. CVEs are demanding applications: real-time interaction and high performance terrain rendering are both very sensitive to latency. The challenge for this thesis is to efficiently stream data out of image archives and allow visualisation clients to access it with minimal latency.

## 1.1 Data Domain: Geospatial Imagery

Geospatial images are an extremely valuable commodity. Around the world there is great interest in earth observation data [45]. All major space organisations include an earth observation group, as do most defence forces and an increasing number of commercial enterprises. Geospatial imagery is valuable because it can be applied to many different fields, including:

- meteorology and weather prediction;
- military command, control, communications and intelligence (C3I) services;
- scientific assessment of environmental impact and climate change;
- town planning and monitoring the effects of urbanisation;
- insurance assessments for underwriting major construction projects.

Estimates vary widely but by 2005 the commercial market for geospatial imagery may be worth as much as \$US2.5 billion<sup>1</sup>. The value to the scientific and defence sectors is immeasurable.

Just as there are many applications of geospatial imagery, so too there are many sources of images. The first civilian remote-sensing satellite, Landsat-1, was launched in 1972. Today there are a great number of different earth observation satellites in orbit around the planet, each providing unique and valuable data. Satellites vary in terms of the number of sensors they carry, the spatial resolution and spectral coverage of each sensor and their orbital characteristics. For example, the French SPOT satellites sit in a low altitude polar orbit and provide results from four spectral channels at a resolution of 10–15 metres over an area 60 kilometres wide. By contrast the Japanese GMS weather satellites sit in a high altitude geo-stationary orbit and provide images which cover a significant portion of the southern hemisphere, but at a resolution of no better than one kilometre. Obviously these differences have a profound affect on the images produced by each satellite. Figures 1.1 and 1.2 show examples of imagery from several different systems.

The diversity of data presents many challenges to those who use geospatial imagery. Perhaps the greatest challenge is due to the sheer volume of data. Individual geospatial images are very large; time series of images can be massive. This means that geospatial imagery is usually held in a mass-storage facility, such as a tape silo. Such devices are optimised to stream out data at high bandwidth, but incur significant initial latency. Other challenges relate to the processing and dissemination of images. Most geospatial data is archived in raw form or with minimal loss-less filtering. Consequently it requires considerable processing before it is fit for direct visualisation. Efficiently processing and streaming large images to visualisation clients is a challenging problem.

## 1.2 Visualisation Domain: Collaborative VE

A Collaborative Virtual Environment (CVE) is an artificial space in which geographically separated people can meet and interact. CVEs have the potential to revolutionise the way we communicate and work because it is possible to see and do things in a computer-generated space that are not practical or possible in the real world. Constraints such as distance, scale, visibility, occlusion, mobility and cost need not apply in a virtual environment. This freedom of interaction has been essential to the success of CVEs in many different domains, including:

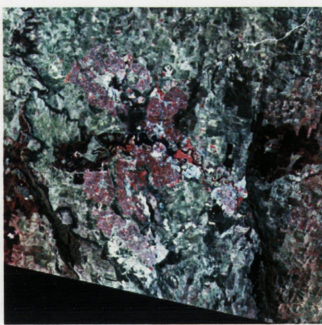
**Scientific Visualisation** – where multiple participants cooperatively explore scientific data sets, or interactively steer a simulation running on a remote High Performance Computing (HPC) resource [108].

---

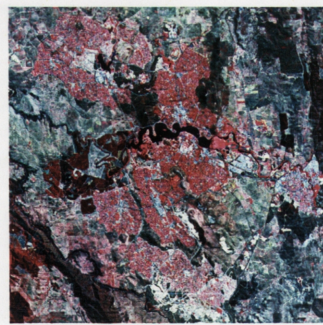
<sup>1</sup>This estimate by Tom Watts of Merrill Lynch is based on the assumption that satellite imagery will replace much of the existing market for aerial photography, currently valued at \$US2.4 billion. Ron Stearn, an analyst with Frost and Sullivan, offers a more conservative figure of \$US420 million if the two markets remain separate.



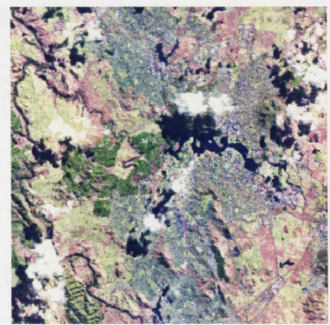
**Figure 1.1:** A composite, false colour image of our planet, produced by NASA.



(i) Landsat MSS



(ii) SPOT



(iii) IRS-1C

**Figure 1.2:** The city of Canberra, as seen by three different earth observation satellites: (i) Landsat MSS, (ii) IRS-1, and (iii) SPOT.

**Military Training and Simulation** – the armed forces have a long history of using computer simulations to train personnel. They have well defined protocols and architectures for building scalable simulations and virtual battlefields, including the Distributed Interactive Simulation (DIS) protocol [68] and the Higher Level Architecture (HLA) [25]

**Cooperative Design** – designers and engineers can work in a shared space to build virtual prototypes rather than physical models. VE-based CAD tools have been developed by a number of large vehicle manufacturers, including General Motors and Caterpillar [83]

**Education** – a Virtual Environment can be an exciting and compelling setting for many education and learning applications [74]. For example, the Narrative Immersive Collaborative Environment (NICE) is a virtual garden, where young children can interact and play with a range of objects and plants, and explore a number of learning themes [119]

**Medical Training** – a particularly specialised and successful use of VEs for educational purposes is in the area of medicine. Collaborative VEs can be used to provide general practitioners with access to specialists [114]. Rich haptic-visual environments are also being developed to allow trainee surgeons to develop their skills.

**Entertainment** – in financial terms video games have now surpassed mainstream cinema and recorded music as the most valuable global media [69]. Multi-player and on-line video games are a particularly high growth genre. They are a unique category of Collaborative Virtual Environment, requiring both a complex set of interactions and a high degree of scalability. Massively multi-player games such as “Ultima Online” and “Everquest” commonly support thousands of players interacting in a shared virtual world [29]. Future games have even more ambitious scalability goals [22].

**Socialisation** – a Virtual Environment can also be an appropriate neutral venue for people to meet and socialise. Multi-user VRML [153, 154] worlds are popular as virtual chat forums and for other types of casual, non-competitive interaction. Users of such worlds often go to a considerable effort to customise their appearance and environment and generally build a rich virtual persona.

The major challenge in developing a Collaborative VE is always responsiveness. Real-time interaction demands a high degree of responsiveness and also requires a constant rate of response [109]. Neither of these characteristics are easily achieved given the latency and jitter inherent in contemporary networks. As a result it is always a major challenge to integrate a new type of data, such as geospatial imagery, into a Collaborative Virtual Environment.

---

## 1.3 The Problem Framed as Four Functional Requirements

The ultimate goal of this thesis is to enable collaborative visualisation of geospatial imagery within Virtual Environments. This is a general goal which encompasses a number of more specific, research issues. Consequently we will frame the problem in terms of four basic, functional requirements:

1. To provide visualisation clients (CVEs) with low latency access to very large geospatial datasets. This we will characterise as a requirement for *responsiveness*.
2. To process and disseminate imagery in a way that makes efficient use of high-bandwidth networks and high performance computing resources. This is a requirement to optimise the rate of *throughput*.
3. To share data required for user interaction and a shared sense of presence. This implies a requirement for *collaborative data sharing*.
4. To support management of resources and configuration of applications in a heterogeneous distributed environment. This is a requirement for *application management*.

These four requirements define the scope of this thesis and the solutions it develops.

### 1.3.1 Responsiveness

Response rate is all important to a Collaborative Virtual Environment. To achieve a sense of real-time interaction visual, auditory and haptic<sup>2</sup> rendering must be performed within stringent time constraints [109]. Visual rendering typically occurs 30 – 60 times per second and, although it can be pipelined to some extent, a lag of even a few frames becomes noticeable and distracting. Bi-directional audio channels are more tolerant, but typically cannot work with latencies greater than 100 – 200 milliseconds. Haptic rendering is most demanding: updates are performed 500 – 1000 times per second affording a latency of one or two milliseconds at most. It is crucial to visualisation to have timely, low latency access to data. In other words, *responsiveness* is a fundamental requirement.

### 1.3.2 Throughput

Low latency data access is not the only performance requirement. The size and processing requirements of geospatial images mean they must be streamed efficiently. Tertiary storage facilities can migrate data off store at very high rates: often faster than a commodity network can handle. This means that high bandwidth networks are required to disseminate images efficiently. Many image processing operations are highly parallel and perform very well on high-performance computing facilities. Consequently it is important to be able to access, transfer and process images at very high rates of *throughput*.

---

<sup>2</sup>Haptics provide force-feedback and a sense of touch within a virtual space.



### 1.3.3 Collaborative Data Sharing

Why is the collaborative aspect important? There are a many reasons why group exploration enhances the way we use geospatial imagery. Earth observation images provide a complex picture of our world. Interpreting them can be difficult and time consuming, and usually requires specialist knowledge. In this context there is considerable value in having multiple analysts develop a collective interpretation of an image. Furthermore, when images are being used as part of a decision making process, it is clearly preferable if all stake-holders can view the images at the same time, sharing observations and forming a consensus. For example, in a military context it may be valuable for officers in the field and trained image analysts to participate when command staff use reconnaissance data to make tactical decisions. There is also an economic argument in favour of collaboration. Access and processing large datasets incurs considerable costs in terms of network and computational resource consumption. These costs can be shared when more than one user views a dataset. Finally there are technical arguments for considering collaboration in this dissertation. Firstly, the information shared for collaboration can also be used to improve application performance. Secondly, collaboration is not easily retro-fitted to existing solutions and should to be built in from the start. So *collaborative data sharing* is a fundamental requirement.

### 1.3.4 Application Management

Large-scale and widely distributed computing systems present a number of unique problems, relating to *application management*. The central issue is how resources are used in a decentralised, heterogeneous computing environment. There are two sides to application management: the need of an individual application to use resources; and the need of a resource owner to moderate use of the resource. On the one hand each distributed application needs answers to the following questions:

- what resources are available?
- when are they available?
- how do I access them?<sup>3</sup>

These questions are all about discovering resources and configuring applications to deal with heterogeneity. The answer is to provide clients with generic means of describing, looking for and accessing resources. On the other hand, resource owners need answers to a different set of questions:

- how do I make clients aware of my resource?
- how do I guarantee availability of the resource to a client? (to avoid over-committing a resource)

---

<sup>3</sup>Use of the word *access* is deliberate. We must assume that an application knows how to *use* a resource once it is made available. The challenge is getting the application to the point where it can use a resource – a configuration problem.

- how do I limit availability of the resource? (to prevent use by inappropriate clients, to stop a single client monopolising the resource, to make the resource available only at certain times)

These are questions of resource management, and are about ensuring that a resource is used appropriately, efficiently and fairly. Answering these questions also requires that clients use standard ways of finding and accessing resources. Viewing geospatial images in a CVE involves a variety of network, data and computing resources. Hence a systematic approach to *application management* is required.

## 1.4 Scope and Suitability of the Problem

In setting the scope of this work the aim has been to consider a problem which is broad enough to be interesting, yet focused enough to be achievable. All work has limits and imagery dissemination and collaborative analysis present challenges beyond the four basic functional requirements described above. Perhaps the most significant is the process of searching for images. This is a rich and complex problem in its own right, and is not considered in this work: rather it is assumed that the user already knows about the images that he or she is interested in. Image searching is an area of active research and, in this respect, a CVE client is no different to any other. Security is another important issue not considered here. For military applications security is paramount and the value of geospatial imagery means that security is also relevant for other applications. Finally, collaboration brings with it a unique set of issues. Although this work is firmly rooted in area of Collaborative Virtual Environments it does not concentrate on the human factors associated with collaboration, but on the underlying problem of data distribution<sup>4</sup>. Important as each of these problems may be, they are beyond the scope of this work.

With any problem where data volume or processing rates are the limit it is reasonable to ask if Moore's Law will ultimately provide a solution. Will the massive increases in storage and processing capacity of future systems fundamentally change the way we address the four functional requirements? The answer is no. We can expect that the percentage of data held in online tape silos will increase. However, just as storage and processing capacity is increasing, so too are the spatial and spectral resolution capabilities of earth observation systems. New hyper-spectral, high resolution satellites are coming online all the time and the volume of data produced by these new systems will be considerably greater than currently available. If anything, the volume of data will increase faster than our ability to deal with it. It is reasonable, therefore, to suppose that visualising geospatial imagery in Collaborative Virtual Environments will remain an interesting research problem for many years to come.

---

<sup>4</sup>Essentially this is a thesis about data sharing in a CVE, rather than the higher level question of how users achieve meaningful interactions with that data. This work is a necessary precursor to that later question.

## 1.5 A Solution

The solution proposed here is a software architecture for imagery dissemination that optimises responsiveness and supports collaborative data sharing. This addresses the runtime performance requirements of a single application. The architecture also addresses the broader application management requirement for a production system. Known as the Responsive Architecture for Pipelined Imagery Dissemination (RAPID), it consists of three main elements:

1. A dissemination pipeline used to access data from imagery archives, process and transform it in arbitrary ways, and deliver it to visualisation clients. The pipeline uses parallel streaming techniques to optimise bulk data movement between processing operations. Parallel streams are formed using a flexible connection object. Data movement is further optimised by flow control and scheduling mechanisms. These all work to meet the basic requirement for high rates of throughput.
2. A parallel data cache, deployed at client sites to optimise responsiveness. The RAPID Cache (RACE) isolates visualisation clients from the latency of the dissemination pipeline. It uses speculative fetching policies and approximates results to further improve responsiveness. In addition it provides a inter-site communications topology to support collaborative data sharing.
3. An application management framework to allow dissemination pipelines to be formed within the context of a computational grid [35]. This includes a well-defined life-cycle for pipelines, a collection of brokers and traders to manage resources and a schema for publishing resource descriptions in a public directory service.

The centrepiece of the architecture is the RAPID Cache, a middle-tier component which performs a unique set of functions. It integrates latency sensitive visualisation clients with a high-bandwidth, high-latency dissemination pipeline without coupling their executions.

## 1.6 Approach and Methodology

This thesis adopts the methodology of a software architect: a practitioner's approach grounded in the development and review of working systems. Independent solutions exist within the data and visualisation domains for each of the four basic functional requirements. The challenge is to integrate these disparate elements into a coherent whole. In this context an architectural approach can bear much fruit.

RAPID is based on the experience of several experimental and real-world systems developed by the author. This experience is presented in three case studies. Highly distributed systems do not always lend themselves to quantified examination. Often the most valuable lessons are learnt from reviewing the architecture of a successful

---

system. Consequently the evaluation of experimental systems is a blend of experimental results and systems analysis. The goal of the experimental work is to distill general architectural principles from practical system.

Where possible, standard notations are used to report results. I deliberately avoid defining my own language, calculus or notation. System diagrams are presented in the Unified Modeling Language (UML) [39]. Use-Cases [71] and Design Patterns [40] are used to characterise system behaviour. This style of reporting reflects a belief that exploratory research should not preclude good software engineering practice.

## 1.7 Organisation

In keeping with the architectural approach, this thesis is organised along the lines of the classic waterfall model of software engineering. It moves from the general to the specific through chapters which consider requirements, specification, design and implementation. Real software engineering projects rarely follow the linear progression implied by the waterfall model, and this thesis is no exception. The transition from abstract to concrete is a useful descriptive device, but does not present the work in chronological order. In fact the experimental systems were all built before the RAPID architecture was specified. So the case studies validate some portions of the architecture, and motivate others.

Chapter 2 reviews formative concepts from relevant literature. The emphasis here is not on an exhaustive taxonomy of research into geospatial imagery or collaborative virtual environments. Rather, the aim is to identify key requirements from the data and visualisation domains and so provide the context for RAPID.

Chapter 3 sketches a general outline of a complete solution. It specifies the key techniques and ideas that need to be encapsulated in the architecture. A concrete design is presented in Chapter 4. This describes the three main elements of RAPID in considerable detail, and is the major result of the thesis.

The remaining chapters present three case studies into working systems. Each system demonstrates different elements of RAPID. Chapter 5 presents the first case study, which considers the responsiveness requirements of a visualisation client. It is based on the Comet terrain exploration tool, which was presented at the tenth plenary meeting of the Committee of Earth Observation Scientists (CEOS). The second case study, presented in chapter 6, describes three different imagery dissemination systems, with an emphasis on achieving high rates of throughput in a distributed pipeline. The final case study, in chapter 7, considers how a computational grid can provide the required application management services. It describes the vGrid, a framework for executing and managing Collaborative Virtual Environments and the resources they consume.

Chapter 8 summarises the major results and describes various directions for future research. One of the many exciting things about this area is that so many questions remain. By developing an general architecture, rather than a specific software toolkit, this thesis aims to provide a foundation for future work.



---

# Formative Concepts

---

This thesis sits at the convergence of two existing areas of research: geospatial imagery and Collaborative Virtual Environments. It is informed by results from both the data and visualisation domains. Before diving into a detailed consideration of how to integrate the two domains, it is useful to review the state of the art in each. There are also several important areas of distributed computing research which are relevant. This chapter considers the major concepts and significant literature that provide the context for RAPID.

## 2.1 Managing and Using Geospatial Imagery

The term *geospatial imagery* is applied to a highly diverse collection of earth observation data. Mature techniques already exist for collecting, managing and processing this data. In this section we will review some of the essential characteristics of earth observation data and the methods currently used for storing and processing it. We will also look at approaches to tiling and degrading imagery, and the inevitable issues of coordinate systems and map projections which arise from using tiling mechanisms.

### 2.1.1 Earth Observation Systems and Data

There are many kinds of geospatial imagery produced by the various satellites and sensors in orbit around the planet. Some of these were mentioned in Section 1.1 of the previous chapter. Although there are many differences in earth observation system it is possible to define a number of characteristics common to all geospatial imagery. Each image may contain more than one viewable channel, representing different portions of the spectrum or different sensors. The major component of each channel is a two-dimensional contiguous block of pixel data<sup>1</sup>. All pixels in a channel will be recorded with the same precision (bits per pixel), which usually im-

---

<sup>1</sup>This pixel data is usually supported by a variety of ephemeral data to describe the conditions, time and orientation of the vehicle when the image was recorded.

plies an optimal packed encoding of a channel<sup>2</sup>. The contiguous nature of the pixel data also implies that tiling strategies can be used to decompose the images into smaller blocks, or to form different levels of detail. Finally, images are recorded on a regular basis and so each individual data set is part of a larger temporal series. Often the frequency of recording is a function of the orbital characteristics of the satellite, and can be quite complex (for polar orbits in particular).

From these characteristics it is apparent that each individual pixel of a geospatial image has extent in the following dimensions:

- two or possibly three spatial dimensions<sup>3</sup>;
- the channel set or spectral domain;
- the temporal dimension.

In addition there will be a collection of ephemeral metadata associated with each image.

Perhaps the defining characteristic of geospatial imagery is simply its size. Individual images from current generation satellites may be tens of megabytes in size. As a result, even short time series of images consume very large amounts storage space. For example the GMS-5 satellite produces 210MB of compressed data per day: approximately 75GB each year. Yet GMS-5 is not a new satellite and by current standards its output is quite modest. Next generation satellites are expected to produce terabytes of data *every day*. Furthermore the number of new earth observation systems is increasing, as commercial companies enter the market. By the end of 2001 companies such as Space Imaging [66] and Orbimage [107] will have launched a series of civilian satellites with resolutions less than one metre<sup>4</sup>. The trend is clear: in coming years we will have access to more geospatial imagery, and at a higher resolution than ever before.

### 2.1.2 Storage and Repositories

Geospatial data consumes great amounts of storage space. Although the size of secondary storage (disks) continues to increase according to Moore's Law, the volume of geospatial imagery is increasing at least as fast. The result: any non-trivial collection of imagery must be maintained on a tertiary storage system, such as a tape silo. These devices can often stream data out at high rates, but incur a heavy start up delay. The number of tape drives within the device also limits the number of images

---

<sup>2</sup>It is important to note that since earth observation systems are rarely restricted to the visible spectrum they do not return pixel data in familiar (red, green, blue) triplets. Mapping results into the visible spectrum is a non-trivial operation and there are many different algorithms used to produce colour and false-colour images.

<sup>3</sup>Three dimensional imagery is produced from stereoscopic pairs of images. Generally speaking such imagery is rare and the overwhelming majority of geospatial imagery has extent in only two spatial dimensions [45]. However, digital elevation models are common. Use of elevation data is considered in more detail in section 2.2.4.

<sup>4</sup>Space Imaging already operate a commercial satellite known as Ikonos. Orbimage have launched their first and second OrbView satellites, with OrbView-3 and OrbView-4 due for launch later in 2001.

---

that can be accessed concurrently. If a silo only has two drives then only two images can be streamed out of the silo concurrently. So access to tape silos, and the archives of data they contain, must be managed carefully.

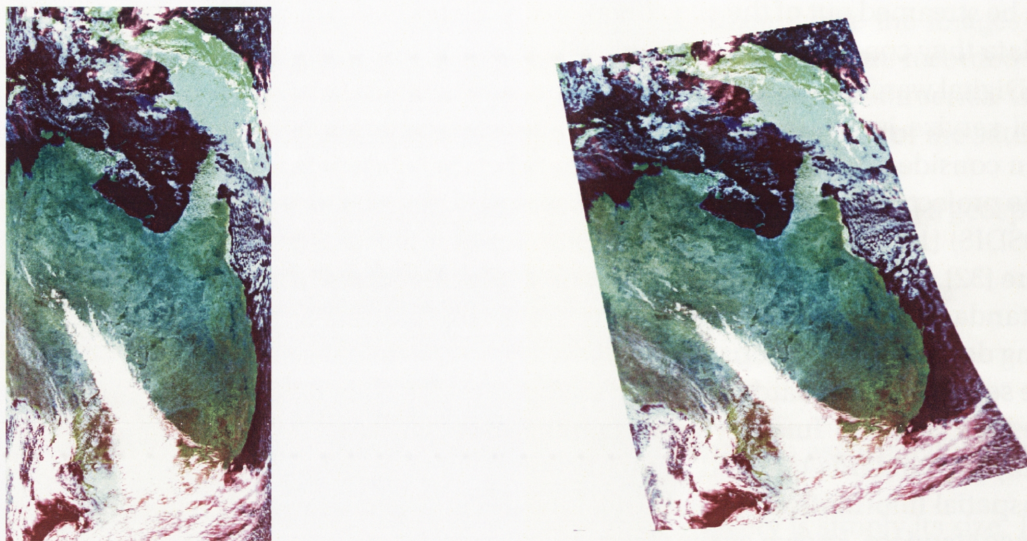
Digital warehouses and libraries have been used for some time to provide a high-level service interface to large tertiary storage facilities [87, 101]. There has also been considerable work providing remote access to archives of geospatial data. Notable projects include NASA's Earth Observing System Data and Information Service (EOSDIS) [110] and the Australian Centre for Remote Sensing (ACRES) Digital Catalogue [32]. These projects were developed in a piecemeal fashion with little thought of standardisation or inter-operability. However, standard archive interfaces are now being developed, with separate initiatives from the defence and commercial sectors. One set of draft standards is being developed by the OpenGIS Consortium [141, 142]. The U.S. National Imagery and Mapping Association (NIMA) has a more mature family of standards targeted at the defence sector, known as the U.S. Imagery and Geospatial Information System (USIGS) [64, 65]. The core of USIGS is an archive interface standard, known as the Geospatial Imagery Access Services (GIAS) [149]. The GIAS is an elaborate specification and developing a full implementation involves considerable work [20, 56].

A significant aspect of all archive interfaces is the way they decouple searching and browsing functions from data access. The GIAS, for example, uses one set of components for query operations and a completely separate set of components for data access [149]. This decoupling stems from an obvious engineering constraint: searching and browsing functions use a catalogue of metadata which is small enough to be maintained on secondary storage, whereas image access involves migrating large volumes of data off tape. The separation is particularly significant to RAPID: it does not consider query issues, and is focused purely on data *access*.

### **2.1.3 Image Processing and Dissemination Pipelines**

Geospatial images require considerable processing before they can be visualised. Most earth observation data is archived in raw form, just as it was received from the satellite. Such data often contains errors and at a minimum requires filtering and cleaning. Rectifying images to correct for curvature of the earth is also a common operation and is computationally expensive to perform. Figure 2.1 provides an example of image rectification. Images may need to be composed and fused if no single image covers an area of interest; which is particularly likely for low-orbit satellites with a narrow field of view. However, building a composite mosaic image from multiple sources is again a complex and computationally demanding process. Finally, since most earth observation systems sample visible and invisible portions of the spectrum there is a need to map imagery into the 24-bit RGB colour values used by display hardware. These operations may be considered the minimum processing required for visualisation. Most users will require additional processing for their particular application. Therefore, a systematic approach must be taken to processing data and delivering it from a storage archive to a visualisation client.





**Figure 2.1:** Rectifying an image to correct for the curvature of the earth. On the left is a raw NOAA image of the eastern half of the Australian continent. On the right is same image rectified to the more familiar Mercator projection. This image processing was performed in close to real-time using a parallel rectification operator running on a high-performance computing cluster.

Many image processing systems adopt a *pipeline* architecture [93]. Pipelines are formed from a collection of processing operators, linked together so that the output of one operator flows into the input of the next. Each operator transforms the data as it flows along the pipeline, with the cumulative results as the output of the last operator. The term pipeline is somewhat misleading since it implies a single sequence of operations. However, many processing operations require more than one input or produce more than one output. For example to map the effects of urban growth involves examining the differences in images taken over a long period of time [45]. When an operator requires more than one input the result is a joint in the pipeline: when it produces more than one output the result is a fork. So rather than being a linear sequence, a pipeline is actually a directed acyclic graph (DAG) of processing operators; otherwise known as a data flow network. This architecture has been widely employed in a great many image processing systems including, AVS [136], the Java Advanced Imaging (JAI) library [134] and PISTON [76, 146].

An alternative approach to image processing comes in the form of active archives [18, 56]. An active archive not only stores data, but also processes it before it is delivered. This allows the archive to produce new products on demand: a form of just-in-time processing. An active archive can also cache the results of computations for use by more than one client. It can even optimise the scheduling and processing of products based on the known behaviour of the tertiary store and the processing tasks [72]. Active archives complement rather than replace pipelined processing ar-

chitectures.

Whether processing is performed within the archive or not, there is still a need to distribute data to visualisation clients. A *dissemination pipeline* is a distributed data flow network which moves data from source archives through a set of processing operations and ultimately delivers it to visualisation clients. An important issue for any data flow network is how and when flows are initiated: this becomes crucial for a dissemination pipeline. A major theme of RAPID is optimising the flow of data along a dissemination pipeline.

#### 2.1.4 Tiling and Resolution

There are two useful techniques for managing the size of geospatial data: decomposition of an image into tiles and degradation of an image to lower levels of detail (LoD). Tiling techniques operate by treating a large image as an array of regularly sized smaller images. Data size is reduced without compromising resolution, but by limiting the spatial extent of each tile. LoD techniques take the opposite approach by degrading the resolution of an image. Hence size is reduced without limiting the spatial extent of an image. Both techniques are valuable for storage management and for interactive visualisation of earth observation data<sup>5</sup>.

Every image can be decomposed into a hierarchy of levels of detail and a three dimensional matrix of tiles of different sizes<sup>6</sup>. These two independent decompositions are demonstrated in Figure 2.2. Using both level of detail and tiling techniques results in quite a complex hierarchy of tiles. This hierarchy is partially depicted in Figure 2.3 as a two dimensional matrix<sup>7</sup>. The visualisation problem is essentially about ensuring that a client has a subset of the tile hierarchy sufficient to satisfy its rendering requirements.

It is often convenient to limit tile size and resolution to be powers of two. In this scheme the largest tile is twice as wide and twice as broad as the next largest tile: the highest level of detail has twice as much resolution, in each dimension, as the next highest level of detail. Note that since we are considering two-dimensional imagery, in both cases the largest tile or highest LoD will be four times the size of the next largest or highest. This organisation is convenient for paging and cache management since it promotes the use of fixed sized buffers of memory.

The two dimensions of the hierarchy shown in Figure 2.3 help illustrate four potentially useful tile operations. *Tile composition* is the act of combining four neighbouring tiles to form a larger tile. *Tile decomposition* is the reverse operation and involves dividing a large tile into four smaller tiles. In the matrix presented in Figure 2.3 composition equates to moving up one level, while decomposition equates to moving down a level. *Tile degradation* is the act of reducing the resolution of a

<sup>5</sup>The requirements of real-time terrain rendering are considered in detail in section 2.2.4.

<sup>6</sup>The three dimensions stem from the two dimensions of the image plus one dimension of tile size. Multi-channel images form a four dimensional matrix, with a fourth dimension for channel selection.

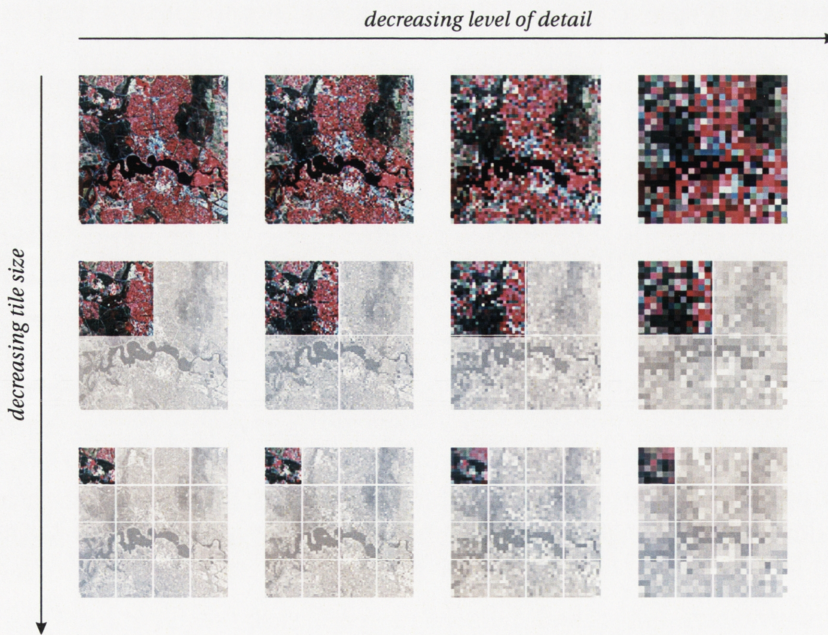
<sup>7</sup>Note that for illustrative purposes Figure 2.3 does not depict the full hierarchy of tiles. Instead it presents the matrix of sizes and resolutions for a single tile



(i) Level of detail hierarchy

(ii) Matrix of tiles and sizes

**Figure 2.2:** Techniques for decomposing images: (i) by level of detail, and (ii) as a matrix of tiles at different sizes



**Figure 2.3:** A partial view of the combined level of detail and tiling hierarchy. This view depicts the level of detail and tile size options available for a single tile in a dataset.

tile, and it corresponds to moving to the right in the matrix. The fourth operation is *tile refinement*, which involves increasing the resolution of a tile and corresponds to moving left in the matrix. For most types of data composing and decomposing tiles is cheap and involves simple concatenation and subset of memory buffers. Degradation is a sub-sampling problem, which may also be cheap for some types of data.

Although tiling and resolution mechanisms are useful for both storage and rendering purposes, it is unlikely that both will use a single common hierarchy. Different image processing operations within a pipeline may also place restrictions on how tiling and LoD are used and on the types and representation of data. One novel solution to this problem comes in the form of the  $k$ -Tile [30] algebra. This provides a way of characterising the arrangement of regular data structures and mechanisms for mapping between different arrangements. It has been successfully used in image processing pipelines to describe and re-map data as it flows through the pipeline [146].

### 2.1.5 Map Projections and Coordinate Systems

An essential part of any tiling strategy is being able to describe what portion of an image any one tile represents. This requires a way to characterise the spatial extent and orientation of both the image and its tiles. Geodesy is the science of measuring and mapping the location of objects on the Earth's surface [27]. It remains an active area of research, due to the irregular shape of the Earth. These irregularities, and our inability to precisely describe the shape of the planet, mean geospatial data is produced using a variety of map projections and coordinate systems. Mapping between coordinate systems, and rectifying images to a different projection, are computationally expensive operations.

Coordinate systems are defined one of two ways: in terms of a two-dimensional map projection overlaid with a reference grid; or in terms of the shape of the earth. The Universal Transverse Mercator<sup>8</sup> (UTM) projections [100] are widely used, but UTM is by no means the only standard for map projection. For example, the State Plane Coordinate System (SPCS) [130] is also common in the United States. Earth-based coordinate systems are characterised as either geocentric or geodetic. Geocentric systems use polar coordinates (angles of latitude and longitude and a distance) from the centre of the planet<sup>9</sup>. Geodetic systems are based on latitude and

---

<sup>8</sup>These are a derivative of the well-known Mercator projection, named after sixteenth century Dutch map maker Gerardus Mercator (1512–1594). The Mercator projection is produced by encompassing the volume of the Earth in a cylinder, where the axis of the cylinder is aligned with the poles and the sides of the cylinder touch the equator. An elegant property of the Mercator projection is that lines of latitude and longitude intersect at an angle of 90°: the disadvantage is that it greatly distorts the polar regions. The Transverse Mercator projection, developed by Johann Lambert (1728–1777), shifts the area of distortion by setting the axis of the cylinder perpendicular to the axis of the earth. UTM is a collection of Transverse Mercator projections formed by dividing the Earth into 60 zones between latitudes 84°N and 80°S, each 6° wide in longitude. There is an extensive body of literature on projections. Useful starting points may be found in [27, 100].

<sup>9</sup>Note that due to the irregular shape of the Earth geocentric latitude is not the same as traditional

longitude and elevation above sea level, but must account for the irregular shape of the world. The World Geodetic System 1984 (WGS-84) [63] defines a standard model of the planet, and is a popular basis for geodetic coordinate systems.

Although the profusion of coordinate systems is inconvenient, it will always be the case that images and coordinates will require mapping. The roughly spherical shape of the planet obviously affects and distorts images as they are acquired, but for visualisation it is usually preferable to treat the surface of the planet as a flat plane. So there is an inevitable mismatch between raw acquisition data, and that required for visualisation. Consequently, the costs of mapping between coordinate systems and of rectifying images are in many ways inherent. This problem is already considered in some standards, such as the OpenGIS Coordinate Transformation Services [143]. Although not considered in detail in this dissertation, RAPID is sensitive to the need for rectification and coordinate mapping.

## 2.2 Challenges with Collaborative Virtual Environments

Having formed an understanding of the data domain it is time to consider how Collaborative Virtual Environments (CVEs) can be used to visualise this data. Collaboration is what distinguishes a CVE from other forms of visualisation, and the importance of shared experience and real-time interaction are hard to overstate. Smith and Weingarten go so far as to describe a CVE as “the ultimate synthesis of networking and media technologies” [128].

CVEs are real-time, distributed applications. With current, best-effort networks there are a number of difficult tradeoffs and limitations which affect how data is distributed; these are reviewed in section 2.2.1. Perhaps the defining problem of CVE research is estimating the position of moving objects in a virtual space; this problem is considered in section 2.2.2. Addressing the basic requirement for collaborative data sharing involves developing loose models of consistency and causality; these are considered in section 2.2.3. Finally, real-time terrain rendering is an interesting and demanding problem in its own right and the major techniques are reviewed in section 2.2.4.

### 2.2.1 Data Distribution Tradeoffs

CVEs make use of a broad range of data types and media, which must be distributed and shared quickly enough to preserve a sense of real-time interaction [138]. Data types commonly used in CVEs include: 3D geometry and models, streams of audio and video media, 2D images and textures, haptic (force feedback) information, results from database queries and output from high-performance computing simulations [24]. These data types each have unique sharing and responsiveness requirements, which translate into a raft of quality of service (QoS) issues. With current

---

latitude marked on most maps.

---

best-effort networks any form of QoS is hard to guarantee, and the solutions that work for one data type may not work for any other. Consequently, despite considerable research into CVEs [15, 25, 68, 84, 97, 145, 153, 159], there is no generic solution to the data distribution problem.

Rather than reveal a single solution, CVE research has identified a range of engineering trade-offs that affect how collaborative data is shared in real-time. In the words of Michael Capps and Chris Greenhalgh:

The field of shared virtual environments has matured. We now have a reasonable understanding of the range of issues that must be considered, and of the key technologies and approaches that can be successfully deployed within particular domains of use. We do not believe that any single technological solution can address every potential application. However, we now have access to a number of enabling technologies that allow us to construct flexible and extensible technology building-blocks, that can be drawn together in different combinations for different applications and requirements.

*Michael Capps and Chris Greenhalgh, CVE 2000*

The *Consistency-Throughput Tradeoff* [126, pages 102–107] is one important engineering compromise that must be made for any CVE. Singhal and Zyda explain the tradeoff succinctly:

It is impossible to allow dynamic shared state to change frequently and guarantee that all hosts simultaneously access identical versions of that state.

*Sandeep Singhal and Michael Zyda [126, page 103]*

Many other tradeoffs exist. In attempting to introduce a new type of data, such as geospatial imagery, into a CVE it is important to understand the factors that will affect how that data is shared.

### 2.2.1.1 Model/View Distinction

In keeping with the Model-View-Controller [77, pages 26–49] design pattern there are two fundamentally separate sets of data used in a CVE. First there is an underlying model of a virtual world. This is the state of the application that must be shared between all participants. Second there is representational data, which is used to render the virtual world to visual displays, auditory systems and haptic devices. For example, a flight simulator must represent the aerodynamic properties, damage and ordinance levels of every aircraft in the simulation as well as having geometry, textures and audio samples to render the aircraft. The former are model data structures, the later are views.

The model/view distinction is significant because the QoS requirements of model data are typically very different to those of view data. In general view data must be updated at very high rates: at least 60 times a second for visual media, roughly 1000 times a second for haptic force-feedback [109, 157]. This makes view data highly sensitive to network latency and jitter. However, the rapid update of view data also makes it tolerant to some unreliability and packet loss: the occasional click or pop in

an audio stream or the occasional dropped frame in a video are acceptable. Model data structures, by comparison, are less affected by latency but do require reliable delivery and high degrees of consistency.

It is not always possible to cleanly separate model and view data, and the distinction is ignored by many CVE toolkits. For example the Avango [145] toolkit completely merges all model and view data into a shared scene graph. While this blurring of the lines simplifies the construction of very simple applications, it does not work for large or complex data such as geospatial imagery. Although, on first impressions, geospatial images may seem to be view structures they cannot be rendered directly<sup>10</sup>. Consequently, RAPID concerns itself with the distribution of model data.

### 2.2.1.2 Data Size

Leigh, Johnson and DeFanti [85] argue that it is useful to categorise CVE data into three sizes: *small*, *medium* and *large*. Small data, in the order of a few kilobytes, can be distributed in real time and broadcast to all participants without too much concern for relevance. A good example of a small data structure is the Entity-State packets<sup>11</sup>, used by the Distributed Interactive Simulation (DIS) [68] protocol to update the position of objects moving in a virtual environment. Medium data is too large for indiscriminate distribution, but still small enough to fit within the memory of an end-user's visualisation machine. Large data exceeds the memory capacity of a typical display machine and so must be maintained on a remote server, partitioned and accessed piece by piece.

Although considerable work has been done on single-user visualisation of large scientific data sets [150], only recently has the problem been considered for a CVE [86]. The goal of RAPID is to integrate one family of large data structures into Collaborative Virtual Environments.

### 2.2.1.3 Access and Communications Patterns

Patterns of access to a data structure have a significant impact on how it must be distributed. In particular the ratio of readers to writers is a crucial consideration [138]. Low-latency read access can be achieved by replicating state to all readers. Write access is much more expensive, since it entails all the problems of locking and serialisation associated with strict equivalence to sequential write consistency [1, 3, 16, 111]. Hence, the number of writers and patterns of access affect how a data structure is distributed.

An intimately related issue is the pattern of communications between hosts in a CVE, and the connections used to access data. When a CVE has well-defined communications patterns it is possible to optimise the movement and routing of data. Such optimisations require the developer to define an application-level topology of

---

<sup>10</sup>Section 2.2.4 considers terrain rendering in detail.

<sup>11</sup>Better known as Entity-State Protocol Data Units (ESPDU).

---

connections, tailored to the particular flows of data between hosts. The first CAVERNsoft [84, 85] toolkit was based on this concept, and affords great flexibility in the way topologies are formed and data shared. CAVERNsoft applications first create a network of connections between hosts, then specify what data should be shared over each connection and how it should be kept consistent. This approach is especially relevant to dissemination pipelines, which have static communication patterns and require optimal throughput.

#### **2.2.1.4 Computational Cost**

Computational cost affects how, and more importantly where, data should be distributed. Rendering a three dimensional virtual environment can easily consume all the cycles of a visualisation workstation. If the environment contains an expensive feature, such as a numerically intensive simulation, then there is a case for centralising the feature on a single server and broadcasting the results to visualisation clients. This approach has been used successfully with many different high-performance computing simulations [13, 24, 108].

Conversely, if there is a low cost associated with a feature it may be better to have replicas on all client machines. Local replicas avoid the latency associated with accessing results from a remote server. They can also be a useful way to overcome communication bottlenecks. If the results of a computation are larger than the inputs, then it is more efficient to share inputs and replicate the computation. Multi-player video games commonly use this technique to overcome the bandwidth limitations of modems [7].

#### **2.2.1.5 Time Sensitivity and Buffering**

Certain types of data are highly dependent on time and/or have a constant rate of change. Such types lend themselves to data streaming. Obvious examples are audio and video media, both of which have constant rates of change and rely on data streaming models for distribution. The significance of time dependent and constant change data is twofold: it is typically very sensitive to variation in network latency (jitter); and it implies a minimum constant bit-rate required of a network. These two issues correspond to quantifiable network quality-of-service requirements. However, time dependent data may be quite tolerant of latency. For example if two users on opposite sides of the world are listening to the same network radio stream it does not matter if one user hears the stream two seconds before the other, what matters is that they both hear the stream at a constant rate.

Buffering is a common technique for data types with stringent timing dependencies [85]. Buffering is used to mask burstiness in the underlying network: it dampens jitter, at the expense of added latency. It is also used to ensure reliable delivery and message ordering over best-effort networks. But buffering is not a panacea: every layer of buffering adds latency. For latency-sensitive applications, such as CVEs, it is important to minimise buffering to avoid unnecessary delays. Considered and



strategic use of buffering is a major element of RAPID.

#### 2.2.1.6 Scalability

Large-scale military simulations involve tens of thousands of participants: current and future multi-player games feature hundreds of thousands of players [22]. Building virtual worlds that can scale to support a very large number of participants is a challenging problem [46]. The solution is to minimise the amount of information sent to each user, known more formally as area-of-interest management.

Area-of-interest management is essentially a load-balancing problem. The simplest solution is to statically partition the virtual space into a number of areas. This approach was used in NPSNET [95, 96] and is effective as long as participants are evenly distributed through the space. Dynamic area of interest management involves creating and managing a collection of interest groups, or locales. Systems such as Mitsubishi Electric Research Labs (MERL) Spline VE [158] and Sony's Community Place [80, 81], as well as numerous multi-player games, have used dynamic interest management with some success. Perhaps the most sophisticated example of interest management was the MASSIVE-2 [47] environment, which was based on a comprehensive spatial model of interaction [5, 46]. These systems reveal a familiar result for dynamic load balancing: there is an important tradeoff between the degree of load balance and the cost of computing the partitioning.

#### 2.2.2 Object Position Estimation

So far the discussion has focused on how arbitrary data types are shared in a CVE. Perhaps the single most important shared data type is object positions, since the movement of objects is a very common cause of change in a virtual space. In fact sharing in real time the position of many moving objects is one of the defining research problem for Collaborative Virtual Environments. Early CVEs, such as SIMNET [75], identified the challenge: it is not practical to simply broadcast the position of an object as it moves. There are numerous problems with a naive broadcast strategy: network latency means that object positions would be spatially correct but temporally incorrect; smooth animation of moving objects would require that rendering and broadcasting be synchronised for all hosts; and the bandwidth use for even moderate numbers of objects would overwhelm a contemporary LAN.

The solution is to use motion prediction to estimate the positions of objects as they move through the space; a technique commonly known as *dead reckoning*. The first dead reckoning algorithms were introduced in SIMNET in 1983, and were so successful they were subsequently used in the U.S. military Distributed Interactive Simulation (DIS) [67] protocol. DIS-style dead reckoning works by broadcasting not only the position of an object, but also the time it was at that position and the characteristics of its movement: velocity, acceleration, rotation and such. This allows a receiver to estimate the future positions of the object. Updates are broadcast only when the difference between the real and predicted positions exceeds a predefined

error threshold. This greatly reduces the volume of data that must be shared, and also allows different hosts to render at different frame rates. Dead reckoning is such an effective technique that it has been used as the basis of many commercial video games, the defence simulation High Level Architecture (HLA) [25], and has been codified as an IEEE standard [68].

Dead reckoning is a useful, general-purpose algorithm for object position estimation but it doesn't guarantee smooth continuous motion. Consequently the basic technique has been improved in various ways to remove discontinuities. The Paradise project [125] was able to refine the technique by using a log of past positions to improve the accuracy of estimates, and converge smoothly when estimated position varied significantly from the real position. It had the added advantage of transmitting less data than classical dead reckoning. Ryan and Sharkey [121] took a different approach by skewing the temporal perception of a virtual space. Rather than attempt to mask the latency of the network, they explicitly modelled it as part of the environment. Data from remote hosts is rendered in different "time zones" depending on the latency between hosts<sup>12</sup>. This approach works very well, but only as long as time zones do not overlap. Finally it is possible to introduce application-specific information to further improve the accuracy of position estimates and at the same time reduce the amount of data shared [78]. For constrained problems special purpose estimators can be built that share the minimum of non-deterministic information about object position [79].

Object position estimation is fundamental to all collaborative virtual environments. Using the criteria defined in section 2.2.1 it is obvious that object positions are a very different type of data structure to geospatial imagery. Dead reckoning packets are small, view-level data structures with minimal computational cost and only limited sensitivity to time. The access and communication patterns are also very different to those of a large geospatial data set. Consequently the data sharing mechanisms for object positions will be different to those used for imagery dissemination.

### 2.2.3 Perceptual Consistency and Collaborative Data Sharing

Interaction between users of a virtual environment requires a common perception of the passage of time, and a shared sense of action and consequence. This is known

---

<sup>12</sup>The idea is best explained in terms of a simple example: two people playing a game of ping-pong, on machines separated by a latency of 200 milliseconds. Both players see the other's position as it was 200 milliseconds ago. The two players are in separate time zones, and as the ball moves back and forth between the players it must move between the two time zones. Consider player 1. When the ball is immediately in front of player 1 it should be rendered in real time: when the ball is immediately in front of player 2 it should be rendered with a delay of 200 milliseconds. Half way between the two players it should be rendered with a delay of 100 milliseconds. By delaying the ball it is possible to achieve perfect spatial accuracy. However, the movement through time also has the affect of distorting the velocity of the ball. As it moves away from player 1 the ball also moves backward in time: the result is that the ball appears to move more slowly than it should. On the way back from player 2 the opposite is true and the ball appears to move faster than it should as it catches up with real time. This distortion is often less disconcerting than errors in position.

as *perceptual consistency*: a much weaker notion of consistency than that applied to classical distributed systems [1, 111]. Perceptual consistency works on the basis that replicas of a data structure are allowed to become inconsistent across multiple machines, so long as these inconsistencies are not perceptible to the end user. Dead reckoning exploits perceptual consistency: the shared sense of an object's position is allowed to become inconsistent but the spatial error is barely perceptible. This form of bounded inconsistency is similar to the notion of Epsilon serializability [112, 165] developed within the database community. However, for an arbitrary data structure it is not possible to define whether an inconsistency is perceptible or not, so it falls to application developers to define the consistency semantics of shared data structures.

Perceptual consistency is a hard concept to support in a general-purpose communications toolkit. The basis of most sequential consistency models is the ability to impose a total ordering on events within a distributed system. The ISIS [8,9] group communication toolkit was used by several early CVEs because it provides total ordering of messages. However, total ordering is expensive to implement and does not scale well. What is needed to implement perceptual consistency is a group communication mechanism that allows the application developer to define what message ordering is sufficient. This is known as *sufficient casual order* among messages.

In practical terms, sufficient causality requires a group communications service with reliable and unreliable delivery modes<sup>13</sup>, and very flexible message ordering semantics. Roberts and Sharkey [117] describe how sufficient causality can be specified by sequencing messages in terms of casual and non-casual order and optionally against a global wall-clock. This approach was used as the basis of the PaRADE [116] and MASSIVE-3/HIVEK [129] toolkits. The importance of sufficient causality is that it provides a very simple programming abstraction, which allows the application developer to minimise the costs of ordering and reliability. It also provides a basis for exploiting imperceptible inconsistency among replicas of a shared data structure.

Provision of a group communication service with sufficient causal order is an important element of RAPID.

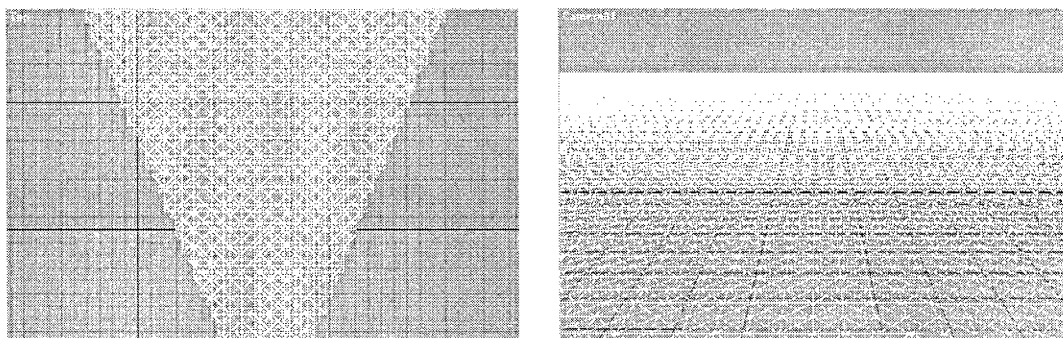
#### 2.2.4 Terrain Rendering

Visualising geospatial data in a CVE involves rendering a three-dimensional terrain surface in real-time. Naive rendering algorithms do not work at interactive rates for even modestly sized pieces of terrain, and so a number of sophisticated techniques have been developed for real-time terrain rendering. A terrain surface is formed from elevation geometry decorated with one or more texture images. This presents two related rendering problems: limiting the number of triangles used to tessellate a terrain surface and managing caches of texture imagery used to decorate the surface.

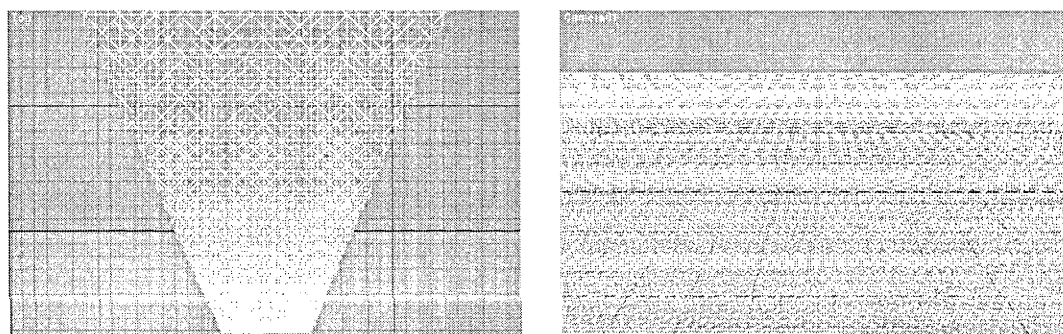
At the heart of all real-time terrain rendering systems is an algorithm to manage level of detail (LoD) in the surface geometry. Such algorithms refine the coarse

---

<sup>13</sup>where unreliable delivery is implicitly assumed to have a lower latency than reliable delivery



(i) uniform surface tessellation



(ii) adaptive surface tessellation with maximum resolution in the camera foreground

**Figure 2.4:** Motivation for adaptive surface tessellation and real-time level of detail algorithms. These four images depict a flat surface which has been tessellated with a constant number of vertices (approximately 15000). Images on the left side show the surface from above and tessellated to the limits of a camera view frustum. Images on the right show the camera's perspective. In the first row (i) a uniform tessellation has been used across the surface. When viewed from above the density of vertices is constant across the surface, but when viewed from the camera's perspective the vertices are concentrated at the horizon. In the second row (ii) an irregular tessellation has been used to concentrate vertices in the foreground of the camera. Viewed from above the surface appears unevenly tessellated, with coarse triangles used furthest from the camera. However, when viewed from the camera's perspective the result is a uniform density of vertices. Adaptive, non-uniform tessellation allows for a more detailed representation of the foreground and avoids aliasing of the horizon. This is the goal of any real-time level of detail algorithm.

LoD techniques described in section 2.1.4<sup>14</sup>. To render natural looking terrain it is necessary to tessellate (decompose) the terrain surface into a mesh of triangles. The mesh need not be uniform in density: some terrain features require many small triangles to represent accurately; others can be approximated by a few large triangles. The location and orientation of the camera is crucial, since it defines the frustum of observable space and so limits the area of the surface which must be rendered. It also defines what resolution different portions of the surface should be rendered at: typically objects in the camera foreground should be rendered at a much higher resolution than those on the horizon. Figure 2.4 demonstrates the inefficiency of a uniform tessellation because most triangles are concentrated on the horizon, where they are of least value. Consequently, the goal of any LoD algorithm is to minimise the number of triangles used to approximate a surface within a frustum of observable space and some bound of spatial error.

Non-uniform surface tessellation can be achieved by subdividing triangles to different degrees. The result is a hierarchy of triangles stored in a quad-tree or equivalent data structure. Early LoD algorithms, such as Lindstrom's [90], used a bottom-up approach to building this hierarchy. Bottom-up algorithms have the disadvantage that they require the elevation data at full resolution in order to build the hierarchy, thus precluding paging of elevation data and so limiting terrain extent. Most current algorithms are derived from ROAM [26], a top-down algorithm which achieves excellent performance using a pre-computed multi-resolution elevation data structure. Perhaps the major driver of current work on LoD algorithms is the computer games industry. Modern games place a premium on photo-realistic graphics and high quality, high performance terrain renders are used in many flight simulators, sports and driving games. Many of these are based on the ROAM algorithm [147] but with considerable modifications to support features such as paging elevation data [148].

Surface tessellation is one major problem for a real-time terrain renderer; the other is texture management. Texture memory in graphics hardware is a very limited resource. MIPmaps [104] are the basic mechanism used by almost all contemporary hardware to manage texture size and optimise use of available texture memory. MIPmapped textures are a hierarchy of two-dimensional images at progressively lower levels of detail. Low levels in the MIPmap are very small and represent the texture at very low resolution; high levels are larger and feature much greater resolution. MIPmapping works well, but only when a texture is small enough to fit in system memory at full resolution. Larger textures, such as geospatial images, require

---

<sup>14</sup>A detailed explanation of the interaction between tiling and rendering LoD algorithms is beyond the scope of this chapter. Essentially, the techniques describe in section 2.1.4 operate at a coarse granularity suitable for paging and streaming data across a network. However, terrain rendering operates at a finer granularity, and to avoid visual artefacts such as tile boundaries and terrain "pop-up" a continuous LoD algorithm is required. The two techniques are quite complementary: a page grain LoD mechanism is used for client-side cache management from which the rendering LoD algorithm derives appropriate visualisation structures. This is an instance of the model-view distinction discussed in section 2.2.1.1: section 2.1.4 describes model LoD mechanisms, this section describes view LoD mechanisms. For more details the interested reader is referred to sources such as [26, 148, 155].

---

a texture paging and caching system. High-end visualisation systems use a hardware based technique known as Clipmapping [124] to page textures in from disk. Software based techniques have also proved effective in many situations, especially when texture data is stored remotely [11, 12, 82].

## 2.3 Distributed Computing

While the data and visualisation domains provide the motivation for this thesis and impose certain constraints, the solution comes in the form of a distributed application. Distributed computing is a broad and mature field, and a detailed review of the literature would be a major work in its own right. This section reviews three fundamental concepts that are essential to subsequent chapters: data streaming, caching and meta-computing. These three concepts are essential to meeting the requirements for throughput, responsiveness and application management.

### 2.3.1 Data Streaming

The stream is one of the most fundamental communication abstractions used today. Streams are an obvious layer to build on top of connection-oriented transport mechanisms such as TCP sockets or ATM virtual circuits [131]. They can be used equally well for structured and unstructured flows of data, and unidirectional and bi-directional flows. Streaming is generally more efficient for bulk data movement than mechanisms such as remote procedure call or remote method invocation [42, 43, 140], because of the straightforward association with the network transport layer. It is also easier to implement network quality of service (QoS) guarantees for streams since they are typically long-lived connections along a single path [28].

Streaming is a particularly effective way of distributing video and audio media. There are numerous proprietary systems for streaming media to web clients<sup>15</sup> and open standards, such as MPEG-4 [102, 113], are also maturing. The limitations of distributed object middleware led the Object Management Group (OMG) to standardise a media streaming system to complement CORBA [52]. There are also a raft of more general network protocols that can be used for streaming, such as RTP<sup>16</sup> [122] and the associated RTSP<sup>17</sup> [123].

One of the reasons streaming works so well for video and audio is because, although the total volume of data is large, the amount required at the client at any moment is relatively small. There is also considerable continuity from one moment to the next. For example, almost all video compression schemes take advantage of the small differences between consecutive frames. The same is true for visualisation problems. Although the dataset may be extremely large, the size and resolution of

---

<sup>15</sup>Examples include the QuickTime from Apple and the Microsoft WindowsMedia tools.

<sup>16</sup>RTP – The Real Time Protocol, developed by the Internet Engineering Taskforce and published in RFC.1889.

<sup>17</sup>RTSP – The Real Time Signalling Protocol, also developed by the IETF and published in RFC.2326.

the screen and orientation of the camera limit the amount that can be displayed in one frame. Furthermore since the camera typically moves along a continuous path, the difference between frames is small.

The other reason streaming works so well for video and audio is because future client requirements are completely predictable. Time-based media has a constant and deterministic rate of change. This is not true for visualisation problems. Since the viewing camera is under direct user control, client requirements are not deterministic.

Streaming provides a basis for solving the throughput problems associated with disseminating geospatial imagery. However, to effectively stream imagery it is necessary to develop client prediction and speculative fetching policies. This requirement is considered in detail in the next chapter.

### 2.3.2 Caching

If streaming is the key to achieving good throughput, then caching is the key to responsiveness. Caching serves two purposes in a distributed system: it replicates data structures and hides network latency [10, page 9]. Replication reduces the amount of data that must be sent across a network backbone<sup>18</sup>. Web caches are a good example of this: most Internet Service Providers (ISPs) ask their clients to access the web through a proxy cache so as to reduce bandwidth consumption [17]. Caching also reduces or masks latency by moving data closer to those who use it. Continuing the web example, clients of an ISP generally accept the use of a proxy cache since the latency of cached pages is much lower than that of other pages. Both properties of caches are invaluable for providing responsive access to large datasets.

The success of any caching strategy depends on two important factors: the cache hit rate, and the cost of a miss. Hit rate is a function of cache size<sup>19</sup> whereas the cost of a miss is a function of the application. Caching is least useful when the hit rate is low since the added latency of using the cache may outweigh the savings on hits. The simplest way to improve hit rate is to increase cache size. The Distributed Parallel Storage System (DPSS) [144] is a good example of this approach. DPSS is a parallel data cache designed to improve the performance of data intensive applications. It is formed from a cluster of machines operating in parallel and so can cache terabytes of data. Parallel caching techniques are described in more detail in the next chapter.

Using large caches is valuable, but even with an infinite cache there will still be a miss the first time an object is accessed. If the cost of a miss is significant then it may be important to predict client requests and pre-fetch objects, thus increasing the hit rate. Significantly, the prediction strategies required to optimise cache performance are the same as those required for good streaming throughput.

---

<sup>18</sup>Obviously this also has a positive affect on throughput, by avoiding sending duplicate requests.

<sup>19</sup>Cache management and replacement policies also affect hit rate. Since these determine how well the cache memory is utilised they can be thought of as essentially affecting the usable size of the cache.

### 2.3.3 Application Management, Metacomputing and Grids

Traditionally, systems which support application management have been known as *metacomputing* environments [127]. At the most basic level a metacomputing environment addresses the twin problems of resource discovery and resource management. A metacomputing environment may also provide support services such as a standard communication mechanism, user authentication and security, clock synchronisation and file sharing.

Resource discovery is a well understood problem, and many mature solutions exist. There are essentially two approaches to resource discovery: broadcast advertisements or centralised registration. The advertisement approach involves each server periodically broadcasting messages to describe the resources it offers. Protocols such as Novell Netware, Java Jini [99] and the Session Description Protocol (SDP) [53] use the advertisement broadcast approach to resource discovery. The alternative approach is to enumerate resources and register them in a public database or namespace. LDAP<sup>20</sup> [60,156], DNS, the Java RMI Registry [133] and the CORBA Naming Service [105] are all designed to support resource registration. There are obvious trade-offs with both approaches. Resource advertisement does not require a dedicated naming service and is very effective in zero configuration/strict-peer environments. However, it does not scale well: as the number of servers increases so does the number of advertisements and at some point the volume of advertisements will become overwhelming. Resource registration can scale more efficiently if the namespace is structured into a hierarchy and portions of the namespace are replicated. In general the advertisement approach works well in LAN environments, but registration is more effective in the wide-area.

Resource management is a more complex problem, since each management policy depends greatly on the resource being managed. In other words the solutions which work for compute resources do not work for network resources or for data resources. Compute resources, such as massively parallel supercomputers, are usually managed on a very coarse grain; the entire machine is allocated one job at a time. Job scheduling systems such as DQS [135] and Condor [92] organise the execution of jobs through elaborate priority queues and provide accounting mechanisms to record who uses a machine and for how long. Network resource management requires a more fine-grain approach and involves allocating resources to support quality of service (QoS) guarantees [28]. In most cases this is achieved by conservative over-provisioning of bandwidth and other resources in order to meet an applications peak demands. Managing data resources requires a different approach again, with the emphasis on user authentication and access control policies. Consequently, a suitably general-purpose resource management system for metacomputing remains an area of active research [72].

Metacomputing has evolved and matured over time. One of the earliest metacomputing systems was the Open Software Foundations Distributed Computing En-

---

<sup>20</sup>LDAP – The Lightweight Directory Access Protocol



vironment (DCE) [118]. DCE provided a comprehensive set of standard services including a structured hierarchical name space/directory, Kerberos security, a time service and a standard remote procedure call (RPC) mechanism. It performed user authentication, allowed federations to be formed, and even provided a replicated, distributed file system. However, DCE was not perfect. It lacked features such as a scheduling system, had some performance problems, was costly to deploy and was widely perceived as proprietary technology. While DCE ultimately failed to gain widespread acceptance, most of the ideas and even some of the implementation have since reappeared in other products.

Most recent developments in metacomputing centre on the concept of a *computational grid* [35]. The term *grid* was coined by Ian Foster and Carl Kesselman:

The word “grid” is chosen by analogy with the electric power grid, which provides pervasive access to power and [...] has had a dramatic impact on human capabilities and society. We believe that by providing pervasive, dependable, consistent, and inexpensive access to advanced computational capabilities, databases, sensors, and people, computational grids will have a similar transforming affect, allowing new classes of applications to emerge.

*Ian Foster and Carl Kesselman, [35, page xix]*

Grids provide a framework for solving application management problems, based largely on the use of a standard, global namespace or directory. Grid toolkits such as Globus [34, 36] and Legion [49, 50] provide general purpose services similar to those offered by DCE. Data-centric grids, such as DISCworld [54, 55], are more tightly focused on the application management problems of a specific domain.

Two of the key drivers for grids are Collaborative Virtual Environments [24] and data intensive applications [101]. Members of the grid community are already working on techniques to support real time visualisation of massive datasets. The DPSS [6, 144] cache, described above, is a good example: it is designed to improve the performance of data intensive applications running within a grid. This makes grids an excellent framework in which to solve the application management problems that arise from geospatial imagery dissemination.

Yet by itself a grid is not a turnkey solution: different classes of application must be integrated into the grid infrastructure. What the grid provides is a layered set of services that can be used to support applications. In a late-published paper Foster, Kesselman and Tuecke [38] identify four layers of abstraction in a grid: the basic *fabric* of the grid consists of machines, processes, data and networks; *connectivity* protocols are provided to support remote access to the fabric; above this sit protocols for treating the different types of fabric as generic *resources*; and finally there is a high level model for managing *collections* of resources. They go on to highlight the need for application and domain-specific services and protocols:

Our goal in describing our Grid architecture is not to provide a complete enumeration of all required protocols (and services, APIs, and SDKs) but rather to identify requirements for general classes of component. The result is an extensible, open architectural structure within which can be placed solutions to key ... [user domain] ... requirements.

---

In this context, one of the goals of this thesis is to demonstrate how imagery dissemination systems can be integrated into a grid.

## 2.4 Summary

This chapter has reviewed the characteristics of geospatial imagery and considered the possibility of using Collaborative Virtual Environments for visualisation. Geospatial datasets are very large and the volume of earth observation data is increasing dramatically. Images are archived in raw form to tertiary storage and access, processing and distribution tasks must be organised carefully to optimise *throughput*. Dissemination pipelines provide a structured way to perform each of these tasks. Tiling and level of detail management techniques can also be applied to manage the volume and flows of data within a pipeline.

A Collaborative Virtual Environment is a powerful visualisation tool. It is also a demanding real-time application which requires a high degree of *responsiveness*. CVEs present a range of data sharing problems, identified in section 2.2.1. While general solutions to *collaborative data sharing* remain illusive, the basic requirement is for a group communication mechanism which supports sufficient causal ordering among messages.

The challenge is to integrate these two separate domains. There are three important approaches to distributed computing that can help to this end: data streaming, caching, and metacomputing. The chapters that follow develop these ideas into a complete architecture for responsive, pipelined imagery dissemination.



---

# Techniques and Approach

---

This chapter presents a set of runtime and support techniques which enable efficient imagery dissemination and collaborative visualisation. These are the raw ingredients of the RAPID architecture. Each individual technique addresses only one or two of the basic functional requirements described in Chapter 1. A number of use-cases later in this chapter illustrate how they can be integrated to provide a holistic solution.

Given a waterfall model of software engineering, the last chapter was essentially a requirements analysis of the problem. The results of that analysis are summarised in Table 3.1. Following that analysis, this chapter specifies the major ideas and techniques used in RAPID. Section 3.1 presents seven different techniques which meet the performance requirements for high *throughput*, *responsiveness* and *collaborative data sharing*. Section 3.2 considers how the final requirement, for *application management*, can be met by a computational grid. These various techniques are brought together in Section 3.3 through a set of use-cases. These present, at the abstract level, a complete solution for responsive imagery dissemination. No solution is perfect, however, and section 3.4 reviews limitations of the approach. Notwithstanding these comments, the techniques presented in this chapter represent the state of the art in imagery dissemination and provide a comprehensive basis for RAPID.

## 3.1 Runtime Techniques

The last chapter identified the value of data streaming and caching for improving rates of throughput and responsiveness. We will now consider seven specific techniques for improving the runtime performance of pipelines, and enabling collaborative data sharing.

### 3.1.1 Pipelined Imagery Dissemination

RAPID solves the basic imagery dissemination problem through the use of data tiling and processing pipelines. Tiling is important because it allows us to decouple image

---

### **Geospatial Imagery**

- As the volume of earth observation data increases we need increasingly more sophisticated approaches to managing data.
- Imagery archives provide the tools to manage data and use separate interfaces to search for images and to access images once found.
- Image processing and data distribution tasks can be organised into a dissemination pipeline.
- Tiling and LoD provide a means of limiting data size, but raise issues of coordinate system and map projections.

### **Collaborative Virtual Environments**

- Geospatial images are very large Model data structures.
- The regular structure of dissemination pipelines allows for optimised communications mechanisms and topologies.
- The computational cost of image processing means it should not be performed on visualisation clients.
- Buffer placement can have a critical impact on responsiveness.
- Scalability in terms of users is not a high priority.
- Object positions are shared over a different topology to imagery.
- Perceptual consistency requires group communications which preserves causal order among messages.
- High-performance terrain rendering algorithms can be adapted to work with page streaming.

### **Distributed Computing**

- Data streams provide a way to optimise throughput.
  - Caching maximises responsiveness and also helps with efficient use of bandwidth.
  - Both streaming and caching require scheduling and speculative fetch policies to be really effective.
  - Computational grids are a metacomputing environment for solving application management problems.
- 

**Table 3.1:** Formative concepts and major results from the literature: the requirements for responsive imagery dissemination

---

processing and data transfer tasks. The essence of pipelining is to overlay the processing and data transfer of one tile from that of another. A dissemination pipeline is formed from a collection of image processing operators, linked together so that data flows from a source image archive to an end-user visualisation clients. This idea is not new. The defining characteristics of the RAPID pipeline are:

**Tiling with Level-of-Detail** – The pipeline has explicit support for tile oriented dissemination at varying levels of detail. Each operator defines what tile sizes and levels of detail it supports for its outputs.

**Request driven** – Tiles do not automatically flow down the pipeline from archives to clients, but must be explicitly requested. In other words the pipeline uses a client-pull model rather than server-push.

**Request prioritisation** – Each tile request has a priority associated with it, where high priority requests are serviced before low priority requests.

**Flow control** – In addition to basic request and response signalling the pipeline supports flow control messages to prevent operators from becoming either overwhelmed with work or starved of work.

**Asynchronous** – Requesting a tile does not cause the requesting process to block. This helps decouple execution in the pipeline; an aid to responsiveness.

**Flexible Streaming and Abstract Connections** – The pipeline must be independent of any one single communications mechanism. An abstract connection object is used to join operators. This can be implemented many different ways to support various protocols and forms of parallel streaming.

**Multiple images per connection** – More than one single image can be shared over each connection between operators. Connections are typed, and all instances shared over a connection must be of the same type.

**Metadata delivered during connection** – Runtime metadata<sup>1</sup> is shared as part of the process of connecting to an operator. This describes the datasets, the tiling models and levels of detail supported by each output of the operator.

**Multi-user** – Collaboration means multiple users can access the pipeline simultaneously. Each operator allows more than one connection to its output ports. Operators also ensure that multiple, simultaneous requests for the same tile are satisfied by a single response.

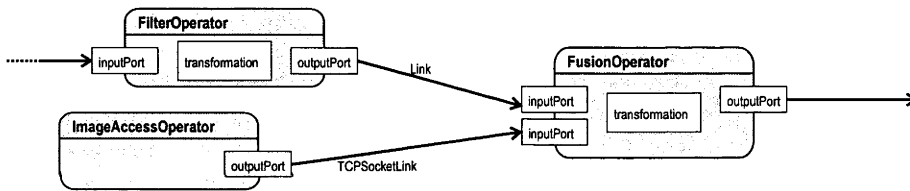
**Buffered** – Results are buffered at each operator to improve multi-user performance when two requests for the same tile occur one immediately after another. This aids responsiveness and throughput by caching data along the pipeline and minimising duplicate requests.

---

<sup>1</sup>Runtime metadata is information which describes the runtime characteristics of the pipeline. This is separate to the application and infrastructure forms of metadata described in section 3.2.

**Operator control and service provision interfaces** – Each operator has separate control and service provision interfaces. The control interface is used to connect clients and for basic service control, and is accessed through a high-level communications mechanism such as Java RMI [133] or CORBA [105]. The provision interface consists of the input and output ports of the operator, joined by abstract connection objects.

A pipeline is a set of connected processing operators. Each operator has a number of input and output ports. Connections are formed between two operators by joining the outputs of one to the inputs of the other. Collectively the inputs and outputs of an operator form the service provision interface to that operator. Movement of data between these ports is performance-critical, so optimised communications mechanisms are used. No one single mechanism is mandated: rather an abstract connection object is defined which may be implemented many different ways. In addition each operator supports a control interface used to connect clients and to manage the execution and life-cycle of the operator. The control interface is not performance-critical and so it may be provided using a single common communications mechanism, such as CORBA.



**Figure 3.1:** Example operators in a dissemination pipeline

To connect two operators a *connection object* must be retrieved from the output port of the upstream operator and sent to the input port of the downstream operator. This object contains the logic necessary to send requests up the pipeline and results back down the pipeline. Different implementations of this object allow different network transports to be used as appropriate<sup>2</sup>. For example a simple TCP socket may suffice to connect two operators running on different machines in the same local area network. However, if the two operators are located on the same machine a more efficient mechanism is to use shared memory buffers. Alternatively, if the operators are separated by a long-haul network it may be more efficient to use a specialised transport mechanism. The abstraction provided by the connection object allows for very efficient communications mechanisms to be exploited. It is used later to introduce forms of parallel streaming into the pipeline.

When connecting two operators a range of configuration details are also passed in the connection object. All ports have a type identifier, possibly in the form of a

<sup>2</sup>Such techniques are often used in implementations of the OMG CORBA to optimise communication performance between objects located on the same machine [44, 151]

---

MIME-style enumeration. Input and Output ports can only be connected if their types match. Configuration information also includes descriptions of the tiling and level-of-detail hierarchy in place, a list of all images exported by the upstream operator, and the metadata associated with each image. To avoid having duplicate connections to a single data source, multiple images may be shared over each connection. Typically each image will have associated with it a range of metadata. At a minimum this will describe the size of the image and the pixel representation used within it, and a pair of texture coordinates which represent the location and size of the image when mapped into a common coordinate system<sup>3</sup>. The metadata may also include details such as the number of channels within the image, the data and time it was taken and the position and orientation of the satellite. Processing operations may choose to add to this metadata, or to filter out irrelevant details.

The request-driven nature of the pipeline is important. There are several reasons why this is not practical to automatically stream tiles down the pipeline in a traditional server-push model. First, a client's requirement for tiles is non-deterministic so operators at the source of the pipeline cannot reliably determine what tiles will be required downstream<sup>4</sup>. Second, since it is assumed that a client does not have enough storage capacity to hold the complete dataset, the client might discard unrequested tiles, wasting bandwidth and throughput. Also, sequencing the execution of operators becomes a problem in a server-push model, especially when an operator has multiple inputs. Many data flow networks require a centralised agent to sequence the execution of operators [146]. Although this approach is acceptable in the intimate environment of a high performance parallel machine, it does not work well in a widely distributed context. Distributed pipelines naturally lend themselves to a pull-driven model as described by the Streams [21, pages 417–426] design pattern.

The RAPID pipeline is request-driven. Clients at the end of the pipeline make explicit requests for tiles to the operator immediately upstream from them. The operator determines what data it needs to satisfy this request and, if necessary, makes a request of its own to the next upstream operator. In this way a chain of requests flows up the pipeline in the reverse direction to the flow of tiles. This provides a natural mechanism for sequencing the execution of operators, and deciding what data should flow down the pipeline and when. Although each operator provides a different set of outputs, all tile requests must contain a common set of attributes. These attributes identify the dataset, channel, level of detail and specific tile required, and are summarised in Table 3.2.

In a collaborative system different users may request the same tile at the same time. This is especially likely when users view features of interest or “hotspots” within a dataset. Each operator is responsible for identifying simultaneous requests for a tile. Rather than have a request propagate up the pipeline more than once, an operator maintains a list of interested clients for each outstanding tile request.

---

<sup>3</sup>Producing this mapping is a whole other problem domain, and beyond the scope of this work.

<sup>4</sup>Partial solutions to this problem are possible, and are considered in section 3.1.2.



- 
- Dataset** – ID of one dataset available from the operator. Since several different datasets may be shared over each connection a request must identify which dataset a tile request is for.
- Channel** – ID of one channel within the dataset. Many geospatial images will consist of multiple bands or channels, so a request must identify a single channel.
- Level of Detail** – Each operator defines a hierarchy of detail from which tiles must be selected.
- Tile** – UV texture coordinates and size of a tile which must be valid within the detail hierarchy defined for the operator.
- 

**Table 3.2:** Attributes required for a tile request

When an operator receives duplicate requests for a tile the operator adds the requester to the list. When a response is available it is sent to all clients on the list. To aid in situations of close, but not simultaneous, requests each operator also buffers a small number of tiles after output. This allows it to respond immediately when client requests are not simultaneous, but do occur in close temporal proximity.

### 3.1.2 Pipeline Scheduling

A request-driven pipeline requires a regular stream of requests to achieve and maintain maximum throughput. Unfortunately, visualisation clients do not usually produce request at a regular rate. Users tend to dwell on and move between features of particular interest within a dataset, resulting in very bursty request patterns. Experimental work in Chapter 5 reveals that even the peaks of these request bursts rarely generate enough traffic to saturate a contemporary network. So a pipeline scheduler is required to ensure that high rates of throughput are achieved in data access, transfer and processing.

The scheduler has a second, equally important, role to play in disseminating imagery: it attempts to mask the latency of the pipeline in the interests of responsiveness. Client request patterns are actually quite predictable. Clients always request a contiguous set of tiles. Client motion through a dataset is usually along a continuous path. Finally, clients tend to focus on particular features of interest within a dataset. Given this predictability, a scheduler can make speculative requests for tiles in anticipation of future client requests. If it makes requests with sufficient lead time it can completely isolate a client from the latency of the pipeline.

The scheduler uses one or more scheduling policies to generate a stream of speculative requests. The aim of scheduling policies is twofold: to maximise throughput of the pipeline, and to mask the latency of the pipeline by anticipating future client requirements. Because scheduler requests are speculative in nature, they are made

---

with a lower priority than that of user demand driven requests, and are satisfied only when there is spare capacity in the pipeline.

Pipeline scheduling policies are either static or dynamic in nature. Static policies are based purely on the known characteristics of a dataset; dynamic policies operate in response to user actions. Although dynamic policies offer potentially greater performance by adapting to changing client request patterns, they also introduce greater complexity and require knowledge of how visualisation clients operate. Four scheduling policies are applicable to real-time imagery dissemination.

**Schedule by Level-of-Detail** – The simplest, static scheduling policy is to fetch an entire image starting at the lowest level of detail and ending at the highest. This is conceptually similar to progressive image downloading used by web browsers. This policy is effective because clients typically start by viewing an entire dataset at low resolution, then request additional resolution for interesting features. It also guarantees that any part of a dataset can be represented temporarily at a low resolution while higher resolution results are fetched. Implementing this policy is very simple: every tile in a dataset is first retrieved at the lowest level of detail before any tile is retrieved at the second lowest of detail.

**Schedule by Request Proximity** – Perhaps the simplest dynamic scheduling policy is to speculatively request tiles adjacent to those already requested by a client. Experimental work in Chapter 5 demonstrates that proximity is a very good heuristic for speculatively fetching data because the client always renders a contiguous area of the dataset. So when a client requests one tile there is a high probability that it will also require the adjoining tiles. Further away from the original request there is still a reasonable chance that the client will ultimately want to view tiles.

**Schedule by Areas of Interest** – Many geospatial datasets contain areas or features of particular significance. The resolution of the entire dataset is tailored to reveal these interesting features in appropriate detail. However, this results in very large amounts of data representing areas of little or no interest. Where areas of interest are known or can be identified it makes sense to arrange those areas for processing before the rest. This results in a static scheduling policy known as Schedule by Areas of Interest.

There are many ways of determining areas of interest within a dataset. One technique is to profile access to each dataset and record what areas are requested at high levels of detail. The hope is that by recording those areas that are viewed in detail by one user it will be possible to optimise the viewing experience of future users. This technique assumes that each dataset will be accessed many times and that users have similar notions of interest. Other techniques avoid these assumptions but require more user involvement in the image management process. For example a user may explicitly identify specific

spatial areas of interest, such as a farmer concerned with the area of her property. Alternatively, feature detection algorithms may be run over an image to automatically detect regions of interest. Military users, for example, may use a feature detection algorithm to detect the wake of a ship or from the periscope of a submarine.

A combination of any of these techniques will produce an interest map for a dataset. Where such maps exist they can be used to prioritise the requesting of tiles through the pipeline.

**Schedule by Motion Prediction** – The location and orientation of a user controlled camera determine what portions of a dataset he or she can view. By predicting the user's motion it is possible to anticipate what portions of the dataset the user will require in the future. Sophisticated movement prediction and position estimation algorithms have been developed for CVEs<sup>5</sup> These algorithms could also be used to make speculative request for tiles in anticipation of future client requirements.

There is obvious potential for combining these policies. Schedule by Level of Detail is really a degenerate case of Schedule by Area of Interest, where the interest map shows all areas of equal interest. Schedule by Proximity is an excellent technique for moderating errors in the accuracy of motion prediction. Finally, when a dataset has multiple areas of interest motion prediction provides a means of choosing between areas. In general the static techniques provide a constant background schedule for tiles while the dynamic techniques provide a means of smoothing between bursts of user driven activity.

Although these policies are all conceptually quite simple, in practice they are not trivial to implement. Subtleties can arise when the spatial extent of tiles and level of detail hierarchies are not independent. Also, in situations where tiles do not support composition, decomposition or degradation the extent and level of detail of tiles must be chosen carefully. There is also the question of where within the pipeline the scheduler is implemented. Motion prediction is perhaps best implemented in the visualisation client since it is best able to anticipate future positions of the user viewpoint. If an Area of Interest map is defined for a dataset it makes sense to store that map in the same imagery archive as the dataset. In multi-user systems scheduling is best performed with global knowledge of system behaviour: i.e. in the actual processing pipeline. RAPID uses a specialised component to perform scheduling on a site-by-site basis.

### 3.1.3 Parallel Caches at Visualisation Sites

Client performance is highly sensitive to latency, and caching is an obvious mechanism to mask latency. It is impractical for a client to cache large portions of a dataset,

---

<sup>5</sup>As described in Section 2.2.2.

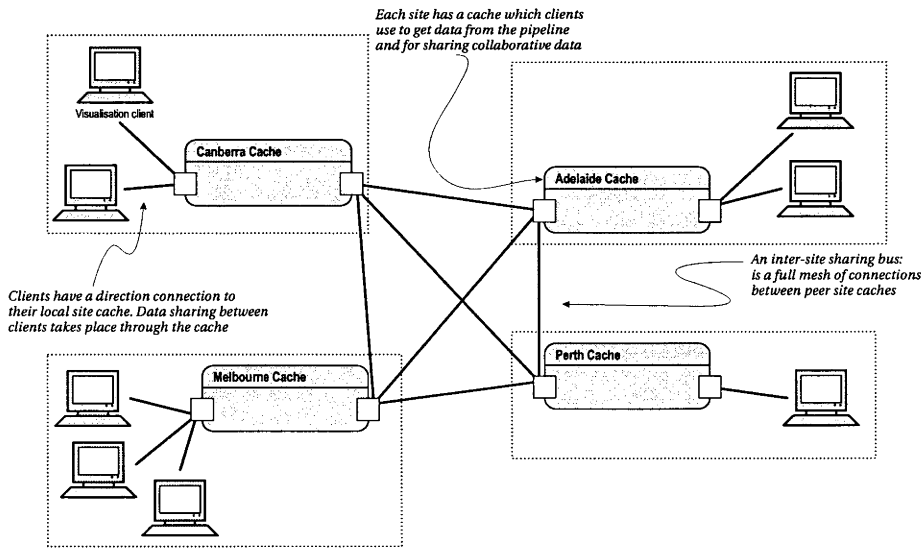
---

but experimental results show that good performance can still be achieved if results are cached on the client's local area network. The use of scheduling policies also has implications for the buffering and caching requirements of the pipeline. A small amount of buffering at each operator is already part of the pipeline model. However, the speculative scheduling policies require a large cache at the end of the processing pipeline to hold results until clients request them. Otherwise the scheduling policies are simply burning valuable network and compute resources.

The solution in RAPID is to provide large caches of tiles located at each site where a visualisation client will run. The definition of a site is deliberately informal. A site might be an organisation unit such as a department or section, or it may be a network centric unit such as a LAN. However two things are assumed about sites. First, that latency between machines within a site is negligible – on the order of a few tens of milliseconds. Second, that there may be more than one user at each site. Each site is connected to the dissemination pipeline through a *site cache*. The cache is attached to the end of the pipeline and acts as a proxy to the pipeline for all clients at the site, forwarding on user requests and caching the results for use by all. The cache also sends speculative requests to the pipeline based on the scheduling policies, described above. This decouples performance-sensitive clients from the latency of the pipeline.

The success of this technique depends on having a low latency connection between client and cache, and on the ability of the cache to hold very large amounts of data. In other words on minimising the cost of cache access and on maximising the number of cache hits. Low latency is achieved by using separate caches at each site. Large capacity is achieved, at low cost, through use of a distributed, in-memory cache design based on the concept of *parallel caching*.

Large site caches can be formed without requiring expensive new hardware by harnessing the unused capacity of machines at the visualisation site. The collective compute, memory and disk storage capacity of a contemporary office environment is considerable, but in normal operation much of this capacity goes unused. Workstation farms use a central scheduling system to coordinate and allocate parallel processes to consume the spare compute capacity of under-used machines. Parallel caches are based on the same idea, exploiting spare virtual memory and disk capacity. Parallel caching has been used in other projects, including the successful Distribute Parallel Storage System (DPSS) [6, 144]. Several things distinguish the RAPID cache (RACE) from earlier systems. RACE is focused specifically on real-time interactive applications and has the same interface as any other operator in a dissemination pipeline. DPSS is a general purpose cache, with a file-oriented interface. RACE also serves to schedule dissemination pipelines and, as described below, is integral to the collaborative data sharing service. While not directly applicable to this thesis, the success of DPSS highlights the value of parallel caching.



**Figure 3.2:** Topology of connections for the inter-site sharing bus.

### 3.1.4 Group Communications for Collaborative Data Sharing

Site caches can be used to implement a group messaging service for collaborative data sharing. Collaborative data sharing affects the operation of the site cache in two ways. First, pipeline request scheduling can be improved by taking advantage of the user movement vectors shared for collaboration (Schedule by Motion Prediction). Second, collaboration requires an efficient topology of connections [84], which complements that of the dissemination the pipeline<sup>6</sup>.

The site caches are an obvious switching point for messages to clients within a site, and for messages between sites. Hence it makes sense to build a group communication mechanism on top of these caches. Establishing connections between caches produces a variation on the *Distributed Peer with Client-Server sub-groupings* topology already used in many Collaborative Virtual Environments [85]. Figure 3.2 illustrates how this topology is formed. Each site cache has connections to all clients at its site, and to all other site caches. In this topology the path between clients at the same site is always two hops, while the path between clients at different sites is three hops<sup>7</sup>. This set of connections is known as the *inter-site sharing bus* and is useful for site cache optimisation as well as collaborative data sharing.

<sup>6</sup>See the discussion at the end of section 2.2.2.

<sup>7</sup>This highlights the principal limitation of this topology: namely that it does not scale well as the number of client sites increases. However, scaling by client sites is not likely to be a priority for any application envisaged in this dissertation. See section 2.2.1.6.

---

### 3.1.5 Inter-site Tile Sharing

Site caches are used to mask the latency of the dissemination pipeline. In the event of a site cache miss, a request will be made to the pipeline that may propagate all the way back to the source data archives. This exposes the latency of the entire pipeline, possibly even including the mass storage device, to the visualisation client. An obvious optimisation, when a tile is not found in a site's cache, is to check the caches at all other sites to see if the tile has already been requested by another user. This allows the collective memory of all caches to be used to mask the pipeline latency. Since the access latencies to remote caches may not be insignificant, the request to other caches can be made in parallel with a request to the pipeline.

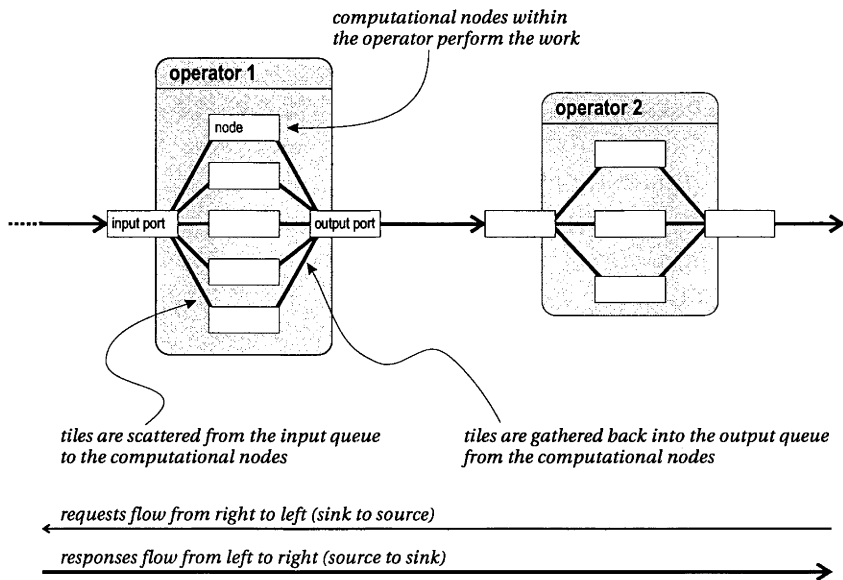
This simple optimisation requires two additions to the pipeline model. First it requires that tile request messages can be made over the inter-site sharing bus, used for collaborative data sharing. Second it may require that requests made of the pipeline by a cache be cancellable. In other words, if a site cache makes a request, then receives a reply from another site, it should be able to cancel the pipeline request so as to save resources.

The importance of the second requirement will vary depending on the relative costs of starting a request and of inter-site communication. Because the latencies between site caches may be non-trivial it makes sense to send a request to the pipeline in parallel with requests to all other site caches. Without this request overlap, a site must wait for a response from all other sites before sending the request on to the pipeline. The danger with this “eager pipeline request” policy is that it may unnecessarily consume pipeline resources for tiles cached at remote sites. If the cost of cancelling requests is very great, and the latency between sites is low, then it may be practical to wait for all sites to respond before making pipeline requests. Hybrid policies are also possible: wait for the first  $k$  sites to respond before forwarding requests to the pipeline; wait  $t$  milliseconds before forwarding requests to the pipeline. In general, a pipeline that supports request cancellation affords the greatest flexibility in site cache design.

### 3.1.6 Parallel Streaming in the Dissemination Pipeline

Many image processing operations naturally exhibit a high degree of parallelism, because pixel filtering is often highly localised. Even complex pixel permutations can be parallelised successfully [146], and there is a wealth of techniques for parallel image processing. Parallelism can also be introduced into the communication mechanisms of the pipeline. Both forms of parallelism can optimise the dissemination of imagery, and help with throughput and responsiveness. This section reviews different forms of parallelism in a processing pipeline and the implications they have on the topology of network connections.

Of itself, pipelining is one simple approach to parallelism. Each stage in a pipeline executes independently and in parallel, with the processing and communication costs of one tile overlapping that of other tiles. However, pipelining has obvious



**Figure 3.3:** Intra-operator parallelism. Two processing operators are shown, each of which consists of several internal computational nodes. Tiles arrive at the operator input queue and are scattered to individual nodes for processing, before being gathered back in the operator output queue.

limitations since the length of the pipeline determines the degree of parallelism.

Higher degrees of computational parallelism can be achieved by executing each operator on multiple computational nodes. This makes an operator a parallel process which receives work on a serialised input queue and sends results to a serialised output queue. Since all parallelism is hidden behind the input and output queues of the operator, we will characterise this as intra-operator parallelism. Figure 3.3 demonstrates intra-operator parallelism for a simple pipeline. Within an operator each tile is handled in three distinct phases: a scatter phase, a transformation phase and a gather phase. Scattering involves assigning a tile from the input queue to a node within the operator. Transformation is the task of actually computing the result and may involve some degree of communication between nodes. Gathering involves collecting results in the output queue.

Image processing operations lend themselves to several forms of intra-operator parallelism. The implementation of any single operation will depend on factors such as tile size, computational complexity, locality and communication requirements. One common approach is to use the Master-Slave [21, pages 133–142] design pattern<sup>8</sup>. In this pattern a single master node manages both input and output queues and is responsible for the scatter and gather phases. Tiles are assigned from the input queue by the master to individual slave nodes for transformation, before being

<sup>8</sup>Also known as Master-Worker [51, page 29] and Worker-Farmer.

---

gathered back at the master in the output queue. This pattern works well for operations with high degrees of locality and limited inter-tile dependencies. It lends itself to implementation on dedicated high performance computing systems where processing nodes do not have direct wide area connectivity, but are instead controlled by a host processor. It also inspires the design of the RAPID Cache (RACE) : where nodes do no actual computation, but simply cache as many tiles as they can.

Parallelism can also be introduced into the communication between operators by removing the operator output or input queues or both, and building additional complexity into the connection object. This is known as *parallel streaming*, or inter-operator parallelism, and is valuable for several reasons. First it removes the serialisation overhead that results from the input and output queues. Second it reduces the number of times a tile must be sent over a network: from three hops to one if both queues are removed. Third it allows multiple data transfers to take place between operators in parallel. Traditional socket based communications are known to perform poorly over high bandwidth, high latency networks<sup>9</sup>. Parallel sockets are often used to improve throughput on such networks. Parallel streaming is also valuable when the bandwidth of the wide area network is greater than that of the local area connection to each computational node. Although no single node can saturate the wide area network, the aggregate bandwidth of several nodes can.

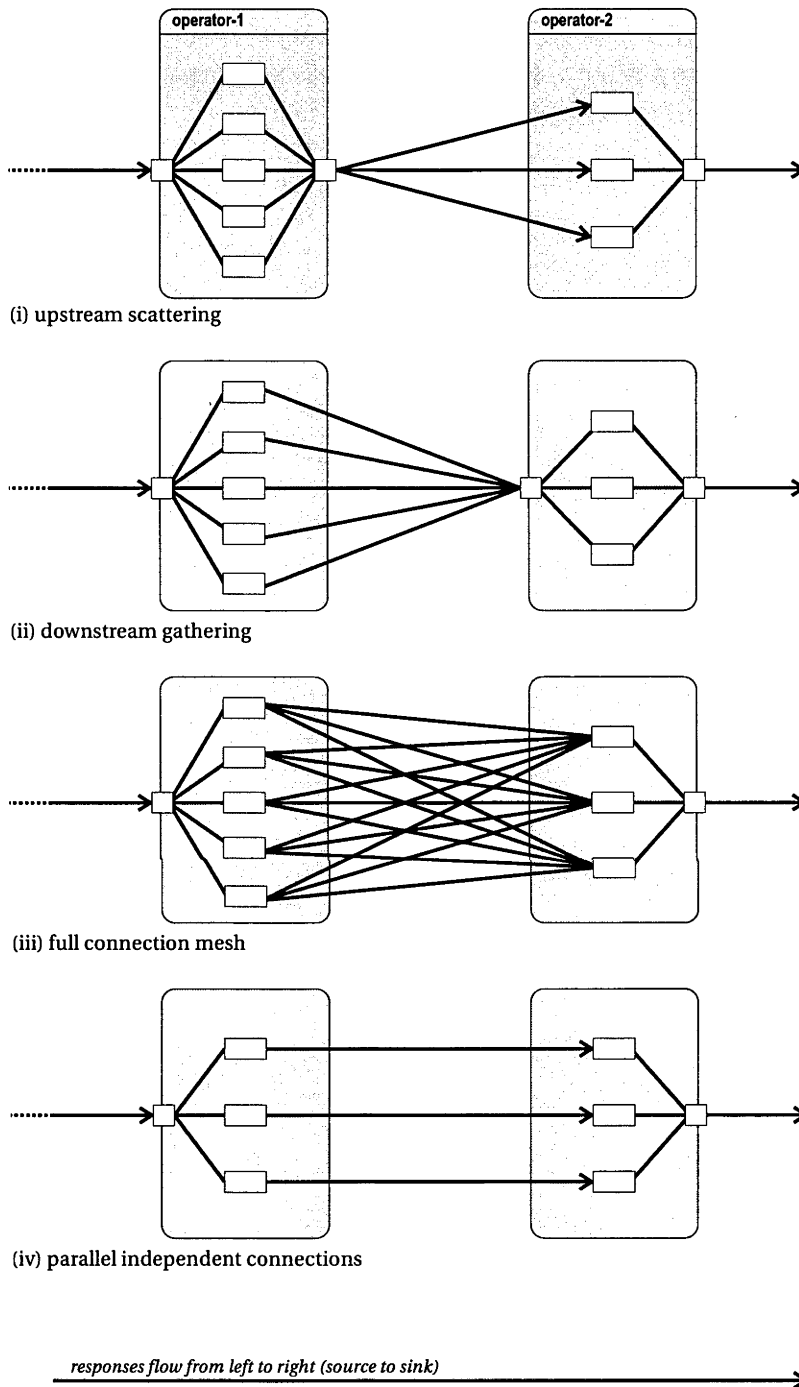
The benefits of parallel streaming come at the cost of greater complexity. Figure 3.4 reviews several forms of parallel streaming. The first form, known as *upstream scattering*, is pictured in Figure 3.4(i) and works by removing the input queue from an operator. Each individual processing node has a direct connection to the output queue of the next operator upstream in the pipeline, and can make independent requests to the upstream operator. Result scattering is performed at the output port of the upstream operator simply by sending a reply along the same connection the request was received from.

The second form of parallel streaming, known as *downstream gathering*, is pictured in Figure 3.4(ii). As the name implies this is the opposite approach to upstream scattering and works by removing the output queue from an operator. The task of gathering is performed at the downstream operator. Each individual node within an operator has a direct connection to the input port of the next operator in the pipeline. Downstream gathering is more complex to implement, because of the need to send requests upstream. Which node does the downstream operator send requests to? The simplest answer is to make a single node in the upstream operator responsible to receiving requests. This node farms the request out to one of the other nodes, Master-Slave style. The more complex answer is to allow requests to be sent to any node, and include a map function in the connection object to select the correct node on a request-by-request basis. A map function could be implemented statically as an HPF distribution policy [59] or a  $k$ -Tiling [30] permutation. Dynamic

---

<sup>9</sup>Principally due to the size of the TCP transmission widow. If the network latency is high then the entire window may be sent before acknowledgements are received at the sender. This has the effect of throttling throughput on high latency links.





**Figure 3.4:** Four models of parallel streaming in a dissemination pipeline. In upstream scattering (i) the results of **operator-1** flow directly to the individual nodes in **operator-2**. Downstream gathering (ii) is the reverse of this, but requires a map function to multiplex requests between upstream connections. A full connection mesh (iii) is a combination of the two approaches but is difficult to construct and maintain. Parallel independent connections (iv) are an interesting special case and may result in a very high throughput pipeline.

---

map functions could be implemented using a portable executable format such as a Java class file or fragment of a scripting language. It is also worth noting that where the mapping function is not entirely deterministic, maintenance of the map will require additional communication in the pipeline.

Downstream gathering is a particularly desirable form of connection between a visualisation client and a site cache. It minimises the network hops required to deliver a tile from the cache to the client. Since this is the most latency critical section of the pipeline downstream gathering minimises response time for cache hits.

The ultimate form of parallel streaming is a combination of the two preceding forms, with both input and output queues removed. This results in a full mesh of connections between operators, as represented in Figure 3.4(iii). Although useful to consider as an extreme case, full connection meshes are expensive and difficult to produce and rapidly become impractical as the number of computational nodes increases. For many applications constraints can be used to limit the number of connections required between operators. A useful special case is that where the number of nodes in all operators is identical, and where each node is connected to only one upstream node. This case is pictured in Figure 3.4(iv) and is effectively parallel, independent pipelines. The result is an extremely high throughput network.

### 3.1.7 Approximating Tiles

Another refinement to the pipeline model is to temporarily approximate the results. If an operator receives a request that it cannot satisfy immediately, an approximation is returned while the precise result is calculated. Approximations could potentially be made through the use of tile composition, decomposition and degradation, as described in section 2.1.4. This allows operators near the end of the pipeline to isolate a client from the full pipeline latency, aiding in client responsiveness.

Imagine a visualisation client connected to the end of a long, slow processing pipeline. When the client requires a tile that has not been processed there will be a long delay while the tile is delivered down the pipe. If an operator near the end of the pipeline can deliver a low resolution approximation of the tile, the client can render the approximation immediately and then substitute the full resolution tile when it is available. Without an approximation the client has nothing to render and the user will see gaps in the dataset.

The use of approximations has numerous implications for the efficiency of the pipeline, the site caches and visualisation clients. In general, approximations are only helpful when both of the following are true:

- when a fast, inaccurate response is better than no response
- when an approximation can be replaced by the correct result without significant side effects.

Both of these conditions are true for a visualisation client at the end of a pipeline, but may not be true for operators near the start of a pipeline. Computing and distributing an approximation is not without cost. The further upstream that approximations

are used the greater the resources of a pipeline they will consume in reaching the client. What is more, the further from the client the less the benefit of approximation. So it may not make sense to approximate results at the start of the pipeline, but may be very helpful in the last few stages before delivery to clients.

Three extensions are required to the pipeline to support approximations:

1. the capability for operators to compute an approximate solution
2. signalling within the pipeline to deliver approximated results when first requested and to replace them with complete results as they become available
3. policies concerning when and how approximations should be used

When approximations are supported there are two classes of result delivered through the pipeline: partial results (PTiles) are temporary, approximated responses to a client request; final results (FTiles) are precise and complete responses to a request. When an operator receives a tile request, the operator must produce a response (FTile) to the request. In most cases the operator will not be able to respond immediately, and will instead propagate a set of dependant requests further up the pipeline. In this situation the original request is said to be *unsupported* by the operator. The request becomes *fully supported* only when complete results are available for all inputs to the operator<sup>10</sup>. At the point the request is fully supported the operator can calculate a complete response (FTile).

Because it may take considerable time for an unsupported request to become fully supported, the notion of *partial support* is introduced to expedite operator responses. This allows an operator to return an approximate or partial response (PTile) to a request as a temporary measure while the correct response moves down the pipeline. A request is said to be partially supported when either a response can be approximated, or when a PTile is available for all dependant requests. The PTiles which form the basis for this partial support are either from upstream operators or are approximated at the local operator. Each time a PTile arrives at an operator the basis for partial support of a request changes. This begs the question, what do we do when multiple approximations are supported? One obvious policy is to produce all possible approximations: another is to produce only the first supported approximation. If the error in an approximation can be estimated then other policies might include: all within error bound; first within error bound; and, ever decreasing error bound. Ultimately, the decision about how best to use approximations is application and operation dependant, but the range of possibilities is clear.

### 3.2 Grid-based Application Management

So far we have only considered how to meet the runtime performance requirements of a single dissemination pipeline. In this section we will focus on the broader questions of resource and application management. Computational grids [35] provide

---

<sup>10</sup>In other words when all upstream operators have responded to the dependant requests

- 
- Operator factory** – A generic service provider, which creates instances of image processing operations.
- Operator** – An instance of an image processing operation produced by a factory following successful service negotiation.
- Imagery archive** – A special type of Operator factory, which provides access to geospatial imagery stored in a repository. This interface complements the main querying interface of an archive used to search for images.
- Image accessor** – An Operator that provides access to one or more images within an archive. On or more image accessors will be at source of all dissemination pipelines.
- Network broker** – A network weather service which reports on the availability and condition of specialist networks.
- Site management agent** – Interface used at client sites to export details of machines, network connections and users into the grid.
- Pipeline trader** – A general service that constructs dissemination pipelines on behalf of clients. This encapsulates the process of service negotiation.
- Pipeline manager** – Produced by a Pipeline trader to manage a single pipeline. The manager is responsible for coordinating provision of service and for allowing visualisation clients to connect to the pipeline.
- 

**Table 3.3:** General classes of actors involved in imagery dissemination

a framework for answering these questions, but by itself a grid is not a turnkey solution. To integrate dissemination pipelines and site caches into a grid we need to define four things:

- the set of basic services, actors and management roles from which pipelines can be formed;
- a resource reservation mechanism;
- a standard schema used to publish descriptions of resources in the grid directory; and
- a standard life-cycle for a dissemination pipeline to bound resource use.

In any pipeline a range of services are required. Table 3.3 summarises the main actors who provide services to a dissemination pipeline. First there are *imagery archives*, which are the sources of raw geospatial imagery and terrain elevation models. These archives produce *access operators* in response to specific requests for imagery. Second there are *operator factories*, which produce processing operators to filter the raw imagery in line with the Abstract Factory [40, pages 87–95] pattern. Third there are *trader* services, which construct dissemination pipelines for a client. In effect the trader performs service negotiation on behalf of a client. Different types of trader produce different types of dissemination pipeline. A pipeline trader produces

a *pipeline manager*, to manage the state of one individual pipeline. Fourth there may be *network brokers*, which provide descriptions of high performance networks and optimised communication routes. Such brokers are common in grid environments and can be very valuable during pipeline construction. Finally there are *site management agents*, which are used by system administrators to publish information about their site, machines, network connectivity and users.

Given this set of actors, a resource schema for responsive imagery dissemination must contain enough information to allow pipelines to be formed and clients to connect to these pipelines. The main purpose of a grid directory is to provide a namespace in which services are advertised. At a minimum this involves associating a service access pointer, such as a Java RMI URL or a stringified CORBA object reference, with a distinguished name. This allows applications to connect to live services that advertise themselves with a well-known name. However, it does not provide any additional metadata to help perform service selection when a client does not know the name of the service it will use. Support for richer means of advertising and finding services is what distinguishes a directory service such as LDAP from a basic name service such as the Java RMI registry or the CORBA Name Service.

The preceding discussion begs the following question: how much metadata should be published about pipeline services? One school of thought within the grid community argues that all metadata should be kept in the directory service. To understand the implications of this argument it is useful to draw a distinction between application metadata and infrastructure metadata. In the words of Chervenak and Foster [18]:

Various types of metadata can be distinguished. It has become common practice to associate with scientific datasets metadata that describes the contents and structure of that data ... We refer to this as *application metadata* ... A second type of metadata is used to describe the fabric of the data grid itself; for example, details about storage systems, such as their capacity and usage policy...

*Chervenak, Foster, Kesselman et al. 2000*

Grids require a directory service to publish infrastructure metadata. The directory often contains the only complete record of the physical and logical components of a grid. However, application metadata is a different issue, and the extent to which it should also be published in the grid directory is not clear. For example, it is inappropriate to publish all the metadata for every image in a large archive: it may be impractical even to list every image. The following principles were applied to the design of the RAPID schema:

1. descriptions of the physical and logical fabric should be published
2. all actors who provide a service should advertise a service brokering interface
3. access to application data should be brokered and the minimum metadata published to let a client decide whether or not to negotiate with the appropriate actor

---

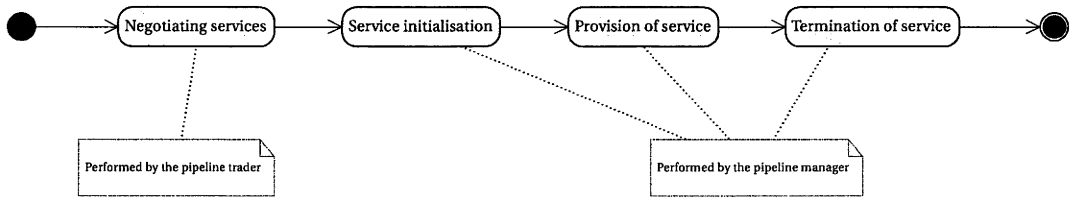
In practice this means where application metadata consists of a few attributes or a short enumeration of options then it may be publishable. Where it consists of an extensive catalogue or is significantly parameterised then access should be brokered through a separate interface. The full schema appears in the next chapter, but in essence it records:

- imagery archive, operator factory, and pipeline trader actors;
- instances of pipelines accessed through a pipeline manager;
- details of the machines and users who may access geospatial imagery; and
- the topologies of high-speed networks which may be used to optimise imagery dissemination.

This brings us to the question of a standard life-cycle for dissemination pipelines. Pipelines consume valuable resources and a standard life-cycle provides a bound on the consumption of resources. The basis of the life-cycle is to separate the negotiation of services from the actual provision of those services. Negotiating for a service is neither performance sensitive nor resource heavy: provision may be both. There are several reasons for drawing this distinction clearly. It provides an opportunity to coordinate and schedule access to resources and so guarantee service characteristics. It provides time between negotiation and provision to prepare a service by staging data off tertiary storage, pre-computing results, or moving bulk data. Finally it allows us to use convenient, high-level communication abstractions (such as ORBs) during service negotiation and efficient, low-level mechanisms during service provision.

The life-cycle of a pipeline really consists of four distinct stages: service negotiation, service initialisation, service provision and termination of service. Figure 3.5 depicts these four stages and the transitions between each. The process starts with a period of service negotiation during which time resources are discovered (through the schema in the directory), and future access is arranged. It may take some time to prepare a service if data has to be staged off tape or if a high-performance computing facility employed. One of the most important aspects of negotiation is to agree on a time at which the service will be available. The initialisation stage can only begin when all required services and resources are available. The dissemination pipeline is formed during the initialisation stage. When all operators are ready, and connections between them have been made, the pipeline is ready to provide data to visualisation clients. This is the stage of service provision. Once clients have finished, or the time limit on resources has expired, the pipeline enters a final stage of termination and clean up. This is the life-cycle all pipelines must follow.

The life-cycle is implemented by two different actors: pipeline *traders* and *managers*. Traders are used to perform the initial negotiation stage on behalf of a client. In effect they encapsulate and solve the resource discovery problem through use of the resource schema. At the end of a successful negotiation the trader creates a new manager, an independent actor in its own right which is registered in the global directory. The manager coordinates the three remaining phases of the pipeline life-



**Figure 3.5:** The life-cycle of a dissemination pipeline

cycle, and also provides a first point of contact for clients when they connect to the pipeline.

The passing of responsibility from trader to manager is made possible through the use of *service tickets*. Resource management is the responsibility of each operator factory. Creating an operator is equivalent to consuming a resource, so factories do not produce an operator while negotiating with the trader. Instead, they provide the trader with a service ticket which contains details of the operator and the time when it will be available. This ticket represents a reservation of resources by the factory, and is a form of Promise [91] to produce the operator at the agreed time. When the trader has finished negotiating with factories it creates a new manager and gives it all the tickets collected during negotiation, along with details of how they should be used to form a pipeline. The manager waits until the time agreed for initialisation, and then evaluates the tickets to produce usable operators.

The final stage in the pipeline life-cycle occurs when clients have finished and the manager must recover all used resources. After the negotiated service period has elapsed, or all clients have disconnected from the pipeline, the manager instructs each operator to delete itself. Any operator may also delete itself if there are no connections to its output ports. This allows for automatic resource recovery of portions of the pipeline in the event of operator failure. The remainder of the pipeline is cleared up by the manager, but only when it has been flushed of all valuable data. This behaviour also allows operator factories to force reclamation of operators if necessary. Such forced reclamations are simply treated as operator failure.

### 3.3 Bringing the Pieces Together

The two preceding sections have described runtime and management techniques relevant to the problem of real-time visualisation of geospatial imagery. Now it is time to bring these pieces together and show how, at an abstract level, they can be combined. To illustrate this process we will use a simple, hypothetical example<sup>11</sup>.

<sup>11</sup>Although hypothetical, the example is representative of the kind of application possible when geospatial imagery and Collaborative Virtual Environments are brought together. A real application might require a more comprehensive set of processing operations, but would otherwise behave the

---

The example provides context for a series of illustrative use-cases, which follow the life-cycle of a pipeline. This provides a comprehensive review of the dissemination process from start to finish.

### 3.3.1 Illustrative Example

Imagine three environmental scientists who use IRS-1 imagery to study the effect of climate change on vegetation levels at several sites within Australia. Joel is an environmental officer with the Department of Agriculture, based in Perth. Markus and Matthew are researchers at the CRC for Biodiversity Measurement, based in Canberra. An important part of their work is monthly reviews of the imagery for each sample site.

During these reviews the three scientists use a common visualisation tool to render a three dimensional view of terrain. The visualisation tool uses a very simple, page-based rendering algorithm, which requires that surface geometry and texture imagery be distributed as one. This requires that the IRS-1 imagery be combined with a tessellated digital terrain model to produce a renderable surface.

The source data for the scientists comes from Southern Cross Imagery, a commercial supplier of earth observation imagery, based in Melbourne. The raw data from the supplier is unsuitable for direct visualisation and must first be run through several processing stages. This consists of fixing errors in the data, rectification of the data and fusing of multiple source images together to achieve complete coverage of the sample sites. The demanding nature of this data processing requires the use of dedicated high performance computing facilities provided by the South Australian Centre for Parallel Computing, in Adelaide.

Each monthly review is organised by Joel. He uses an browser client to select appropriate data sets from the archives of Southern Cross Imagery. Having identified the data sets for a review, Joel tells his browser what day and time to schedule the review, provides details of his collaborators (Markus and Matthew) and describes what imaging operations should be applied to the data in order to visualise it<sup>12</sup>. This is the starting point we assume before a real-time collaborative session can be organised.

### 3.3.2 Negotiating a Service

The first task is to negotiate use of services for accessing, processing and distributing the data. A pipeline trader performs this task on Joel's behalf. The trader advertises itself within the grid directory service. Once Joel has adequately described the requirements of the monthly review, his browser searches the directory service to find a trader that can meet these requirements. The exact process by which a browser selects a trader is not important: it could be an automatic selection based on some set

---

same way.

<sup>12</sup>This last item is a directed, acyclic graph that describes both the set of operations and the connectivity between operators. Encoding of such a graph could be done variously, but XML [164] is an obvious solution



of constraints or heuristics, or it could be a choice displayed in the user interface of the browser. Having selected a trader, the browser passes on Joel's requirements and waits for a response. Obviously the trader may not be able to satisfy all requirements: it may not know how to satisfy some requirements, and there may not be appropriate resources to satisfy others. In either case the trader will inform the browser about those requirements that it cannot satisfy. It is a matter for the browser to decide how best to respond to this situation, but in general this will probably require further user interaction to decide on alternative data sets, or processing operations to apply.

The objective for the trader is to construct a pipeline manager which contains service tickets for all resources required in the pipeline. Joel's requirements include a list of source IRS-1 images that must be accessed, a graph of processing operations to be performed and a digital terrain model over which the results should be draped. Figure 3.6 illustrates a pipeline that would meet Joel's requirements. It contains various operators to access the source data from Southern Cross Imagery, fix errors, perform rectification, tessellate the digital terrain model and fuse all results. The outputs of the fusion operator are fed to tile caches at the end-user sites in both Canberra and Perth, from where Joel, Matthew and Markus' visualisation clients access them. Each operator in this pipeline, including the site caches, is the product of an operator factory. So to construct this pipeline the trader uses Joel's requirements to search for operator factories advertised in the grid directory service and then negotiate with each factory to produce an appropriate operator.

The selection of operator factories is based on various concerns. The type of operation to be performed is the most obvious, but the choice of operator factory could also be based on hints provided by the user. For example, Joel might stipulate that operator factories provided by the South Australian Centre for Parallel Computing are to be favoured above all others. Network connectivity, published in the directory by network brokers, may also be used to inform the choice of factory. Having made the decision to use rectification and tessellation factories in Adelaide, it would be very foolish if the trader decided to use an American factory for the fusion operator<sup>13</sup>. Other concerns might arise in commercial or military contexts. In a commercial system, cost minimisation might be a priority during service negotiation: in a military context security would be a key concern.

However a factory is selected by the trader its role is the same: to produce an imagery operator at the time specified by Joel. The trader and factory must negotiate details of the operator to be produced, and some of this negotiation may even be reflected back to Joel. For example, the factory for tessellation operators may negotiate about the number of computational nodes used within an operator. If the negotiation concludes successfully, the factory provides the trader with a service ticket for the tessellation operator. This service ticket acts as a promise for the operator and guarantees that the factory will reserve an appropriate number of compute nodes

---

<sup>13</sup>This description implies a general routing problem which is again beyond the scope of this work. It suffices to observe that the pipeline routing problem is solvable with existing techniques, and that there already exists a body of work on application-level routing within a computational grid [35, pages 165, 275].

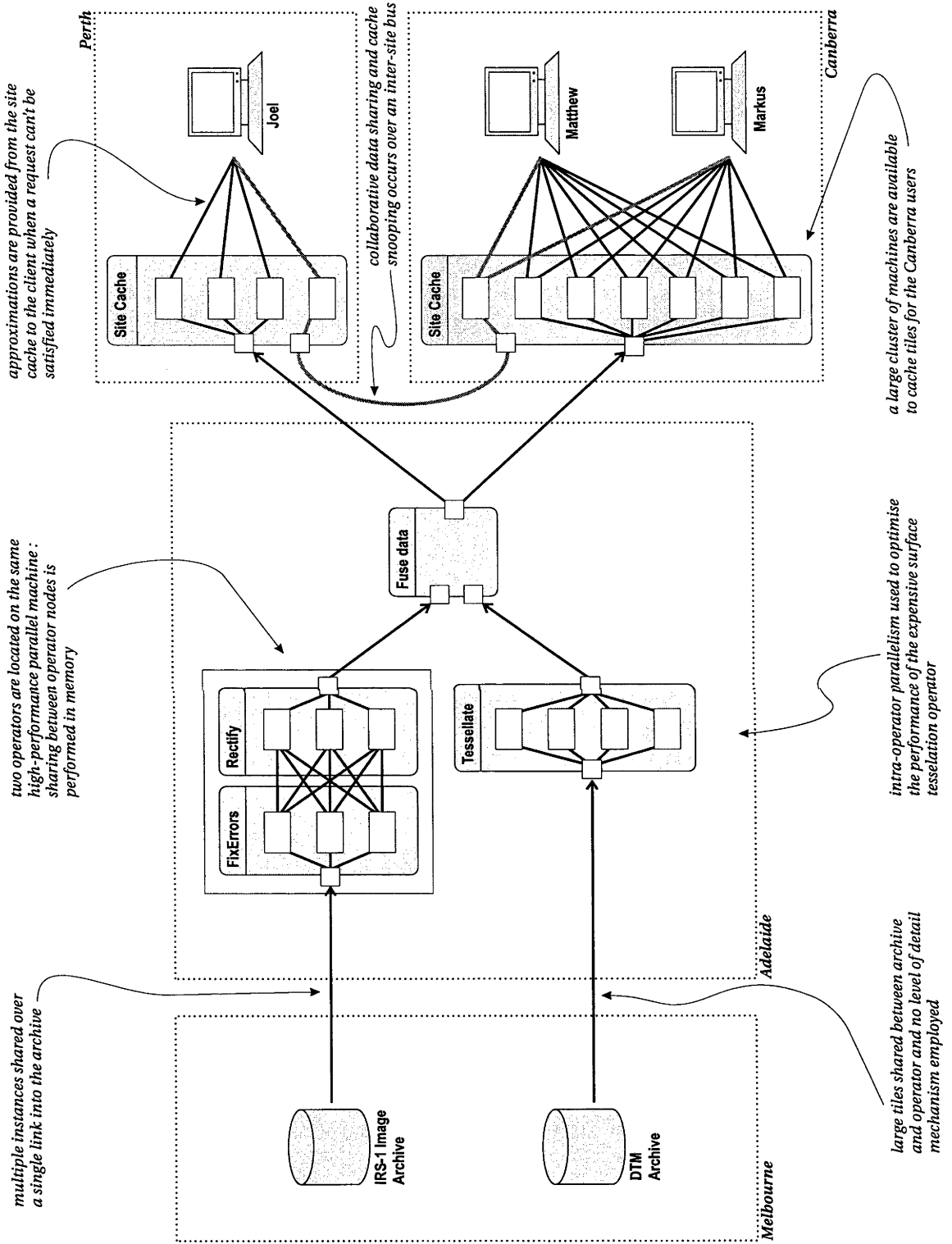


Figure 3.6: A dissemination pipeline for the example application

for use during the initialisation and provision phases of the pipeline life-cycle. Similar negotiations are performed with the Southern Cross Imagery archive to produce data access operators, and with the factories for error correction, rectification and fusion operators. Each negotiation will result in a service ticket for an operator.

When the trader has successfully negotiated production of all the required operators, it creates a pipeline manager. The pipeline manager is responsible for initialising the pipeline at the time agreed for service provision, and for cleaning up after the pipeline is finished. To do this it holds all the service tickets negotiated by the trader, along with the connectivity graph describing the relationships between operators. The site caches are operators within the pipeline, but have particular significance as the connection points for client applications – at the time of service provision it is these operators that clients will need to know about. The pipeline manager also records the user details of Joel, Matthew and Markus. In a secure environment these could be used to limit access to the pipeline: in a highly collaborative environment they may be advertised to attract others interested in the project. Once the pipeline manager is created it becomes an independent actor in its own right, and is advertised in the grid directory service. At this point the trader has completed its task, and returns details of the pipeline manager to Joel's browser. This concludes the service negotiation phase of execution.

### **3.3.3 Initialisation of Services**

The next significant event occurs at the time agreed for provision of service, when the pipeline manager initialises all operators. This will take place before the time Joel, Matthew and Markus intend to view their data. The pipeline manager prepares each operator, starting with the two Southern Cross Imagery access operators and moving through the operator connectivity graph in breadth-first order to the site caches. For each operator the manager first evaluates the service ticket of the operator. This causes the relevant operator factory to make good on its promise of a resource, and instantiate the operator. Second, the manager connects the input ports of the operator to the output ports of upstream operators, as required by the connectivity graph. It then instructs the operator to initialise itself, after which the operator should be ready to produce output.

To create the Image Rectification operator in our example, the pipeline manager starts with a service ticket containing a reference to the Rectify operator factory, and an encoded service ID. When the manager evaluates this ticket a request is sent to the factory, which passes in the service ID and returns with a reference to a newly produced Rectify operator. In the example this operator consists of three computational nodes, and runs on the same parallel machine as the Fix Errors operator.

Next the manager attempts to connect the input port of the Rectify operator to the output port of the Fix Errors operator. It does this by retrieving a connection object from the Fix operator's output port and passing that connection to the input port of the Rectify operator. The Rectify operator essentially runs as three separate processes in parallel. It uses upstream scattering to avoid having a centralised in-

---

put queue, by copying the connection object to all three computational nodes. An added subtlety to the example arises because the Fix operator supports downstream gathering. The connection object passed to the Rectify operator contains a request mapping function that shares work between the three nodes of the Fix operator. The connection is also smart enough to detect that the source and destination are located on the same machine, and optimise data transfer by using shared memory buffers. Consequently, the communication between Fix and Rectify operators uses highly optimised, point-to-point, in-memory messaging.

Having established the connectivity between Fix and Rectify operations the manager initialises the Rectify operator. To initialise itself the operator examines the metadata for each dataset exported from the Fix operator, and performs any other set-up computations. At this point the rectify operator is initialised and ready to service requests for rectified, error-corrected tiles of IRS-1 imagery.

This process is repeated down the length of the pipeline, starting with data sources and concluding with the site caches. There is one subtlety to the initialisation of the site caches due to the creation of the inter-site sharing bus. This bus is used for cache snooping and also to share collaboration data between sites. The bus is formed from point-to-point links between sites. When the manager initialises a site cache it sends details of all caches that will attach to the pipeline. Part of the site initialisation is to establish connections with all other sites. However, this requires that the manager has evaluated the service tickets of all site caches before it attempts to initialise any of the caches. In our example the manager initialises the Perth cache first. As part of its initialisation process the Perth cache opens a socket connection to the inter-site sharing port of the Canberra cache. When, in turn, the Canberra cache is initialised it will already have a connection to the Perth cache in place.

### **3.3.4 Provision of Services**

When the pipeline is fully initialised, services may be provided to visualisation clients. Clients access the pipeline through their local site cache, but the initial connection is made through the pipeline manager. Before the review commences Joel sends the name of the pipeline manager in an email message to Matthew and Markus. The name of the pipeline is a human readable string, and is really a distinguished name within the grid directory service. When Matthew's client connects to the pipeline, it uses this string to look up a reference to the pipeline manager in the grid directory service. It then contacts the pipeline manager and presents it with Matthew's credentials. The manager returns a reference to the Canberra site cache. The client contacts the cache for a connection object with which to make tile requests and receive results.

Having retrieved a connection object to the site cache, Matthew's client connects to the cache in the same way any new operator joins the pipeline. One significant characteristic of the connection is that the site caches use downstream gathering to minimise the cost of delivering a tile to the client. Consequently Matthew's client will have a direct link to every node in the Canberra site cache, with the connection

object providing a mapping function to multiplex tile requests between links. This minimises the number of messages required to service a site cache hit, and keeps cache access latency to a minimum. If the Canberra site cache implements a dynamic load-balancing mechanism, then it will have to send updates to the mapping function to Matthew and Markus.

Once Matthew's client is connected to the pipeline, it makes requests for tiles of data. The terrain rendering algorithm uses the position and orientation of the view frustum, controlled by Matthew, to determine what portion of the terrain is visible. This leads to a series of requests for tiles to represent different portions of the terrain surface at different levels of detail. Each tile request is handled in the same way: we will now examine the sequence of events for one tile  $T$ . Matthew's client first checks its local cache to see if  $T$  has already been received. If  $T$  is missing from the client cache it sends a high priority request to the Canberra site cache. Insuring that the request is sent to the appropriate node within the site cache is the job of the connection object's map function. If  $T$  is available in the Canberra site cache then it is returned to Matthew's client. However, if  $T$  is not in the cache a request for  $T$  is sent to the Fuse Data operator, running in Adelaide. While the Canberra site cache waits for a response to this request it can do two things to expedite the rendering of  $T$  on Matthew's machine. First, if a lower resolution version of  $T$  is available in the cache it can return this as a partial response to the client's request. This allows the client to render an approximation of  $T$  while the full tile is calculated. Second, if Joel has already viewed  $T$  it may be in the Perth cache. So, in parallel with the request to the Fuse Data operator, the Canberra cache can ask the Perth cache if it has a copy of  $T$  through the inter-site sharing bus. If  $T$  is in the Perth cache it can be returned to Matthew by way of the Canberra cache, and the request to the Fuse operator cancelled. If  $T$  is not present in the Perth cache then there is no alternative but to wait for the Fuse operator to respond. The Fuse operator in turn will quite likely make requests to the Rectify and Tessellate operators. So the original request for  $T$  may trigger a sequence of requests back up the pipeline all the way to the source data archives.

When a client request propagates all the way back to the source archive there may be a noticeable delay before a renderable tile is delivered to the client. In a bid to avoid these delays the site caches attempt to predict the future requirements of clients at each site. Prediction generates a set of low-priority, speculative requests for tiles. These requests will vary from site to site. Initially the caches will request a base resolution copy of all tiles in the data set. The Canberra cache may request a higher base resolution than those of the Perth cache, since it has more nodes on which to store results. Following Matthew's request for tile  $T$  the Canberra site may decide to speculatively request the neighbouring tiles  $T^{north}$ ,  $T^{south}$ ,  $T^{east}$  and  $T^{west}$ . It may also use the prior positions and orientations  $p$  of Matthew's camera to extrapolate the position and orientation in five seconds time  $p'$ , then make speculative requests based on  $p'$ . Finally, Matthew and Markus may have defined the location of their sample sites as areas of particular interest, allowing the Canberra site cache to prioritise fetching of those areas.

---

The position and orientation of each users camera is one of the key data structures that must be shared for collaborative purposes. This allows users to see one another and understand what portions of the terrain surface each is viewing. Since the site cache requires this information for speculative fetching, it makes sense that collaborative information is routed through the cache. As Matthew moves his camera over the terrain surface, his client sends camera movement vectors to the Canberra site cache. The cache uses these vectors for speculative fetching, but also forwards them to Markus' client and to the Perth cache (over the inter-site sharing bus) from where they are forwarded to Joel's client. Other collaborative messages follow a similar path. When Joel sends a text-chat message to his collaborators it is sent from his client to the Perth cache, then to the Canberra cache and finally to Markus and Matthew's clients.

### 3.3.5 Disposal of Services

Cleaning up after the application finishes is relatively simple, since all resource dependencies are acyclic<sup>14</sup>. Clean up occurs either when there are no clients connected to the pipeline, or when the service time limit runs out. Assuming no node or network failures have occurred, the pipeline manager shuts down operators one by one, in the reverse order to that in which they were created. The behaviour of an operator when the manager instructs it to shut down is deliberately not specified. Most operators will inform the factory that created them, free any resources they have used and close all connections to upstream operators. When all operators have been cleaned up, the pipeline manager removes itself from the directory service and exits. Failure semantics are discussed in the next chapter, but it is worth noting that resources will all eventually be released in the event that any of the operators, clients or even the pipeline manager fails.

## 3.4 Caveats

The preceding example illustrates the basic approach to meeting requirements for responsiveness, throughput, collaborative data sharing and application management. While complete at the conceptual level, in practice there are limits to any solution. Some limitations, such as the mapping of datasets to a common coordinate system, have already been identified as beyond the scope of this work. There is nothing in the preceding discussion that would limit how these tasks are performed, and in most cases solutions already exist.

Other limitations are less easily dismissed. Some operations have fixed latencies associated with them, which may be too great to be masked by caching, speculative fetching and approximation. An implicit assumption of the pipeline life-cycle is that negotiation is performed well in advance of service provision. In particular, it is

---

<sup>14</sup>Ignoring the inter-site sharing bus which does not consume managed resources.

assumed that the time between negotiation and provision is greater than any fixed, start-up latencies associated with providing the service. In general a user cannot simply request access to a geospatial image and expect it to be available immediately. For example, the task of staging raw datasets off a mass tertiary store is often very slow. Tape silos have certain hard delays associated with moving robot arms, loading tapes into drives, seeking through tapes and streaming data onto fast secondary disk storage. There is no way to short-circuit these delays. A sensible image repository would stage data off tertiary storage well before the time agreed to access an image. However, if an image has not been staged off tertiary storage at the time a user wants to access it, there is very little that can be done to isolate the user from staging delays. The same limitation applies to some processing operations where on-demand processing may not be practical and pre-processing is the only realistic solution. In both cases the only way to provide real-time access to such data is if users identify their intent to view well in advance.

A final and obvious limitation is the assumption that the owners and administrators of image archives and developers of image processing operations are prepared to support new access mechanisms. Without this assumption it is virtually impossible to make meaningful progress on the problem, yet in reality most administrators and developers need strong incentives to adopt any new technology. This is ultimately a social, economic and political problem<sup>15</sup>. It is fair to observe, however, that such non-technical problems remain one of the main reasons that access to geospatial imagery is not more ubiquitous.

### **3.5 Summary**

This chapter has identified a range of techniques which meet the performance requirements for throughput and responsiveness. These centre on a dissemination pipeline which supports parallel streaming and a novel, parallel cache which runs at each site, scheduling the pipeline and speculatively fetches tiles. The site caches are also the basis of a group communication mechanism for collaborative data sharing and cache snooping.

The chapter has also considered how to integrate dissemination pipelines into a computational grid for the purposes of application management. A resource schema published in the grid directory is used to advertise services and for resource discovery. The schema is maintained by a small set of actors, including pipeline traders and managers. A standard life-cycle was defined for pipelines, with a strong separation between service negotiation and service provision. This separation is made possible through the use of portable service tickets which act as resource reservations.

A set of use-cases were used to illustrate how these techniques can be integrated at an abstract level. It is the work of subsequent chapters to make this abstract solu-

---

<sup>15</sup>This could easily be the subject of a separate dissertation – though perhaps not in Computer Science!

---

tion concrete. The next chapter codifies the techniques into a single design: RAPID. Subsequent chapters demonstrate implementations of the design.





---

# RAPID: Responsive Architecture for Pipelined Imagery Dissemination

---

Having considered the problem at an abstract level, this chapter describes in detail an architecture for responsive pipelined imagery dissemination, known as RAPID. It uses the techniques outlined in the last chapter to address the fundamental performance and management requirements. RAPID is a general design, an architecture, not a toolkit or a library. As such it is a design-time solution to the problem: implementations of this design are presented in the case studies that follow.

RAPID decomposes into three major elements: a processing pipeline which disseminates imagery at high rates of *throughput*; a site cache which improves *responsiveness* and also supports *collaborative data sharing*; and an *application management* framework for use in a computational grid.

The structure of the chapter is based on the life-cycle of a pipeline. First, conventions used to report the architecture are described in section 4.1. Section 4.2 outlines the application management infrastructure, used during the negotiation of services. Section 4.3 describes how pipelines are constructed during service initialisation. Section 4.4 describes in detail the operators that form a dissemination pipeline, and the interaction between operators during the provision phase. Operators are built from several sophisticated components, and support parallel processing and parallel streaming. Finally, section 4.5 describes the RAPID Cache (RACE). A RACE is deployed at each visualisation site and decouples clients from the dissemination pipeline. It brings together all the different techniques described in the last chapter, and is the centrepiece of the architecture.

## 4.1 Documenting An Architecture

RAPID is a blueprint, from which a family of systems can be built. A specification of the architecture was presented in the last chapter, and is summarised in Table 4.1. This chapter provides a descriptive review of RAPID, the components from which it is formed and the interactions between these components. To complement it, a comprehensive class reference is included in Appendix A. While reading this chap-

---

**Throughput**

- Parallel streaming optimises the flow of large tile messages down the pipeline, and the flow of request messages from clients to site caches.
- Intelligent connection objects are used to join operators.
- Scheduling policies speculatively fetch tiles and consume unused capacity in the pipeline.
- Flow control manages the rate of requests made to the pipeline and ensures no operator is starved of, or saturated with, requests.

**Responsiveness**

- Large tile caches at each site provide low latency access to data.
- Parallel streaming minimises the response time for cached tiles.
- Asynchronous interactions in the pipeline, ensure that client rendering is not affected by delays in the pipeline.
- Approximations provide a temporary result for non-cached tiles.
- Speculative requests are made to pre-empt future client requests.

**Collaborative Data Sharing**

- Low-latency, one-to-many message delivery between clients is supported with sufficient causal ordering of messages.
- Efficient, concurrent access to the dissemination pipeline is possible for multiple clients at multiple sites.

**Application Management**

- A set of standard actors are used in pipeline construction and management.
  - All service negotiation uses a single, common interface.
  - A well-defined schema is used to describe and advertise resources in a computational grid.
  - The standard pipeline life-cycle decouples service negotiation from provision, and bounds resource use.
- 

**Table 4.1:** Important ideas and techniques: a short specification of RAPID

---

ter it may be valuable to refer to the appendix from time to time, to clarify the roles and interactions of classes. The RAPID blueprint is not totally prescriptive. Like all good architectures it leaves unspecified that which is not central to the problem. Before diving into a detailed description of RAPID we will briefly consider the notations used to report and to identify implementation specific details.

#### 4.1.1 UML and Perspectives

This chapter makes extensive use of the Unified Modelling Language (UML) [106]. Although it is assumed the reader is familiar with UML, a brief review of the notation is provided in Appendix B. UML diagrams can be used to convey varying levels of detail in a design: from the very general to the very specific. The idea of *design perspectives*, developed by Martin Fowler [39], captures the different uses of UML. Fowler identifies three different perspectives from which a design diagram can be viewed: as a conceptual map; as a specification; or as an implementation. The conceptual perspective is most abstract and is independent of implementation details and language semantics. The implementation perspective is concrete, and can often be translated straight into code. Between the two sits the specification perspective, where components and interactions are specified but without the exhaustive detail required for implementation. Specification describes interfaces rather than final classes. Not unnaturally then, the choice of perspective has greatest impact on UML Structure and Class diagrams:

Understanding perspective is crucial to both drawing and reading class diagrams. Unfortunately, the lines between the perspectives are not sharp, and most modelers do not take care to get their perspective sorted out when they are drawing.

*Martin Fowler, [39, pages 51–52]*

The diagrams in this chapter describe RAPID from the specification perspective. Some implementation detail is provided for particularly important objects, and in a very few cases peripheral ideas are left at the conceptual level. Important details that must be provided for an implementation of RAPID are flagged for inclusion in an implementation profile. The rest of the design is a language-neutral, object-oriented description of a highly responsive imagery dissemination pipeline, with appropriate support mechanisms.

#### 4.1.2 Implementation Profiles

RAPID is a design for a family of dissemination systems. There are many implementation details unique to a particular type of geospatial imagery, or image processing operation. Such implementation-specific details are not specified here, but in a separate *implementation profile*. The use of a separate implementation profile is common in many software architectures. In this case it is modelled after the Geospatial Imagery Access Specification (GIAS) [149]. GIAS defines a common interface to image archives, but leaves many imagery specific details to the system implementer.

RAPID defines an architecture for imagery dissemination, but again leaves some details to the system implementer. For example, it assumes all imagery can be registered in a common coordinate system but does not specify what that system is. Although it is specifically designed to support tiling and level of detail mechanisms, it does not mandate any particular mechanism. These details are not essential to the architecture, or the requirements it addresses. They are, however, essential to any implementation. So, like the GIAS, RAPID requires an implementation profile to specify those details that are application dependent.

### 4.1.3 The UML Object Model and New Stereotypes

UML does not provide rich mechanisms for describing the distribution of components in RAPID. Nor does it adequately capture the semantics of objects that move between hosts, or the threads of control in a system. This chapter uses three class *stereotypes* to overcome these limitations: «remote», «mobile» and «active».

The «remote» stereotype is used to indicate a class of object that acts as independent, distributed components. These are typically used during service negotiation and initialisation and are not performance sensitive. They should be implemented using a distributed object middleware, such as CORBA [105] or Java RMI [133]. Indeed the name of the stereotype is deliberately intended to evoke association with the `Remote` interface used by Java RMI objects. The essential capabilities of «remote» objects are that they are remotely accessible, that they do not move between hosts but that references to them are portable between hosts.

The «mobile» stereotype is used to indicate a class of object that can be moved transparently between hosts. Mobile objects are an important element of RAPID. The various messages used in the pipeline protocol are a trivial example of mobile objects. A more important example is the connection object used to join operators. Mobile objects have two major capabilities. First, the state of an object (its fields) can be serialised to a neutral format and shared between machines in a heterogeneous environment. Second, the executable code for the object (its methods) can also be shared between machines. The first capability can be provided by technologies such as XML, Java serialisation or use of the `Serializer` [98, pages 292–312] design pattern. The second capability can be realised through the use of portable executable formats such as scripting languages or Java class files, mobile agents [19, 94, 120], or a dial-a-protocol system such as Bamboo [159, 160].

The «active» stereotype is used to describe how and where threads are used in the design. It takes its name from the `Active Object` [152, pages 483-500] design pattern, and identifies objects which execute in a separate thread. Active objects are a relatively high level abstraction for concurrent programming. Method invocation on an active object takes place asynchronously: usually through a message queue. This abstraction may not be appropriate for all implementations. The «active» stereotype serves as a design-time descriptive aid, but is not a prescriptive implementation requirement. Objects marked with the «active» stereotype identify opportunities for concurrency, which may or may not be exploited in any implementation.

---

Objects that are not marked with any of these stereotypes are assumed to behave as simple run-time objects, and are neither remotely accessible nor mobile and may be called by any thread of control.

## 4.2 Service Negotiation and Application Management

A premise of the RAPID architecture is that dissemination pipelines run within a computational grid [35]. The grid is used to address the application management requirement, and is central to the negotiation of services and the creation of pipelines. As such, it is an obvious starting point for a review of the architecture.

Application management is about controlling the way resources are used in a widely distributed system. The important resources for RAPID are geospatial images, computing facilities to perform image processing, high bandwidth networks and site caches (RACE). So RAPID supports construction of dissemination pipelines from these resources, within a computational grid. It does this by providing answers to the following questions:

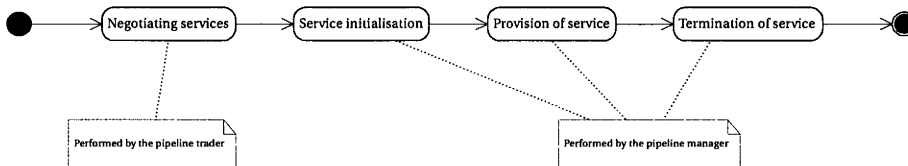
- *What is a resource?* – All resources used in a pipeline are modelled as operators, even high speed networks. An `Operator` represents a single use of a resource: the `OperatorFactory` manages all access to the resource.
- *What resources are available?* – All resources are described by a common schema published in the grid's directory service.
- *Where are resources located?* – An efficient dissemination pipeline is one that minimise the movement of data over wide area networks. To help construction of efficient pipelines the resource schema also describes the location of operators and the connectivity available between them.
- *How do I access a resource?* – A common `ServiceNegotiation` interface is used by all the different types of `OperatorFactory`.
- *When is a resource used?* – The pipeline life-cycle, described in the previous chapter, bounds resource use and allows service guarantees to be made. The life-cycle of each pipeline is governed by a `PipelineManager`.

Answers to these questions come from a well-defined model of service and resource use based on the pipeline life-cycle. RAPID defines a collection of standard actors and objects through which services are provided and resources managed. It also defines a resource schema for advertising services, and a standard interface to resource managers.

### 4.2.1 Actors and the Pipeline Life-cycle

PipelineTrader	Reserves resources for a pipeline during negotiation	p. 169
PipelineManager	Constructs the pipeline during service initialisation	p. 168
OperatorFactory	Manages access to a resource and produces operators	p. 168
Operator	A single use of a resource and one stage in a pipeline	p. 173

Dissemination pipelines all follow a standard life-cycle consisting of discrete phases of service negotiation, initialisation, use and termination<sup>1</sup>. The RAPID management framework is designed around this life-cycle, with a strong separation drawn between service negotiation and service provision. All performance sensitive tasks are confined to the provision phase. The life-cycle is implemented through four main actors: PipelineTrader, PipelineManager, OperatorFactory and Operator. Figure 4.1 depicts relationships between these actors, with several specialisations of the factory and operator.



Visualisation clients do not interact directly with operator factories or the grid support infrastructure. Instead, they use a PipelineTrader to negotiate on their behalf. Each trader can build one or more forms of dissemination pipeline<sup>2</sup>. It uses the schema, described below, to discover resources (and their associated OperatorFactory) advertised in the grid directory. It negotiates with each factory to make appropriate resource reservations and, once all required resources have been found, creates a PipelineManager. The manager is responsible for actually constructing the pipeline during the initialisation phase, connecting clients to the pipeline during service provision, and for cleaning up after the pipeline has terminated.

Individual operations within a pipeline are constructed by an OperatorFactory. There are four major subclasses of factory, with associated types of operator. RACEFactory is a producer of RACE caches at sites where visualisation clients run. ImageryArchive is an abstract class representing an archive of geospatial imagery, and produces ImageAccessor operators to retrieve specific datasets from the archive. A Filter is a general class for any kind of image processing operation, and is produced by a FilterFactory. All real image processing operations are descendants of filter. Finally, Transporter and TransportFactory are specialist network components and are described in more detail in section 4.2.3.

<sup>1</sup>For more details refer to section 3.2.

<sup>2</sup>No general interface is specified for the trader since the form of the interface and complexity of interaction between client and trader will vary greatly depending on the type of pipeline it constructs. Defining a trader interface is an important part of an implementation profile.

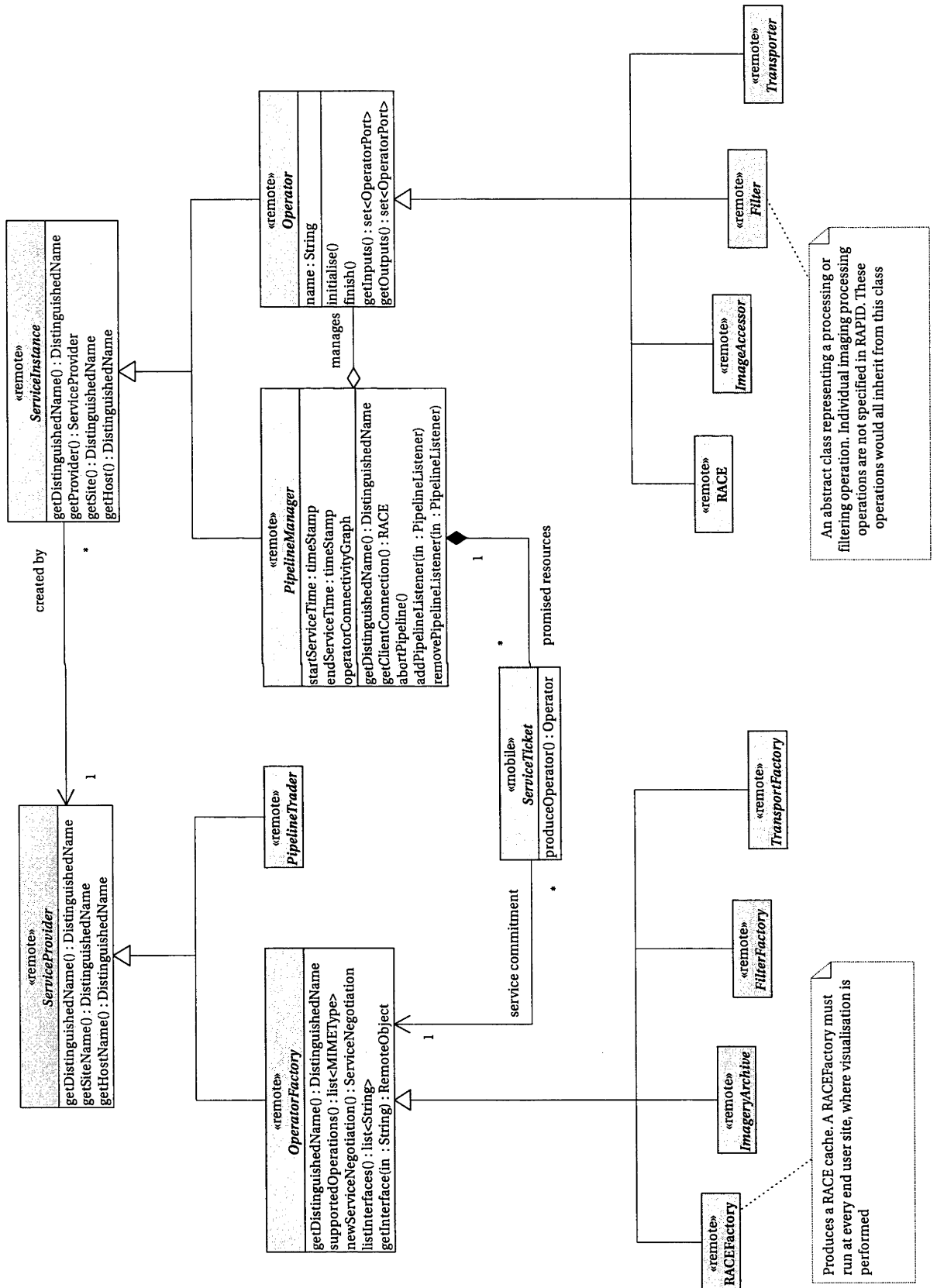


Figure 4.1: Actors involved in application management



#### 4.2.2 Negotiating Access to a Resource

---

<code>ServiceNegotiation</code>	The standard interface for traders to negotiate with factories	p. 170
<code>ServiceTicket</code>	The result of a successful negotiation: a resource reservation and a promise to build an operator	p. 170

---

A `PipelineTrader` undertakes service negotiation for a client. It interacts with factories through a standard `ServiceNegotiation` interface to make resource reservations. The result of a successful negotiation is a `ServiceTicket`, which is both a resource reservation and a Promise [91] by a factory to produce an operator. These tickets are later used by the `PipelineManager` to construct the pipeline during service initialisation.

The `ServiceNegotiation` interface used by traders is accessed through the factory `newServiceNegotiation()` method. Each negotiation object encapsulates the parameters of service for a single operator (i.e. for a single use of a resource). Table 4.2 lists the methods in the negotiation interface, while Figure 4.2 depicts the sequence of interactions involved in a simple service negotiation. The set of service parameters that must be negotiated, will vary greatly from factory to factory. For example, negotiations with a high performance computing facility involve parameters such as the number of computational nodes and amount of scratch disk space, whereas the quality of service parameters for a broadband network might concern minimum, average and peak guaranteed bit rates.

The negotiation object is modelled as a collection of service properties, rather like a `JavaBean` [115]. There are only three standard service properties: the type of operator required and starting and ending times for use of the operator. Operator types are specified using a MIME-style enumeration, which should be defined in the implementation profile. The range of types supported by a factory can be determined by the `supportedOperations()` method. All other negotiable service properties will vary from factory to factory. The negotiation interface provides methods to list the names of service properties, and to distinguish between required and optional properties. Each property has a type, and the list of all types is another important detail that is specified in the implementation profile. There are methods for getting and setting the value of different service properties, and a range of exceptions associated with service properties. A negotiation may only be completed once a value has been provided for all required properties.

When a negotiation is completed successfully, the result is a `ServiceTicket`. A ticket is a promise for a resource: an operator is the fulfilment of a promise. So the result of service negotiation is a lazy-evaluation primitive, which can be evaluated at the time of service initialisation. For a high performance computing facility the ticket is an agreement to run a particular image processing filter at a particular time<sup>3</sup>;

---

<sup>3</sup>An implementation of `ServiceTicket` would obviously have to interact with the site batch queuing system. It may well be an abstraction for an entry in the queue. Computational grids can provide added support for this problem. For some grids it may be possible to implement a single `ServiceTicket` that can be used for all resources across the grid.

---



---

```

setOperation( in: MIMType ) throws ServiceTypeException
setStartTime( in: timestamp ) throws AvailabilityException
setEndTime( in: timestamp ) throws AvailabilityException

listProperties() : list<PropertyName>
listRequiredProperties() : list<PropertyName>
listOptionalProperties() : list<PropertyName>

propertyDescription( in: PropertyName ) : String,
    throws UnknownPropertyException
propertyType( in: PropertyName ) : PropertyType,
    throws UnknownPropertyException
propertyRange( in: PropertyName, out: min, out: max),
    throws UnknownPropertyException
propertyValues( in: PropertyName ) enumeration<PropertyValue>,
    throws UnknownPropertyException

getProperty( in: PropertyName) : PropertyValue,
    throws UnknownPropertyException
setProperty( in: PropertyName, in: PropertyValue),
    throws UnknownPropertyException PropertyValueException

isAcceptable() : boolean
complete() : ServiceTicket, throws ServiceException
abort()

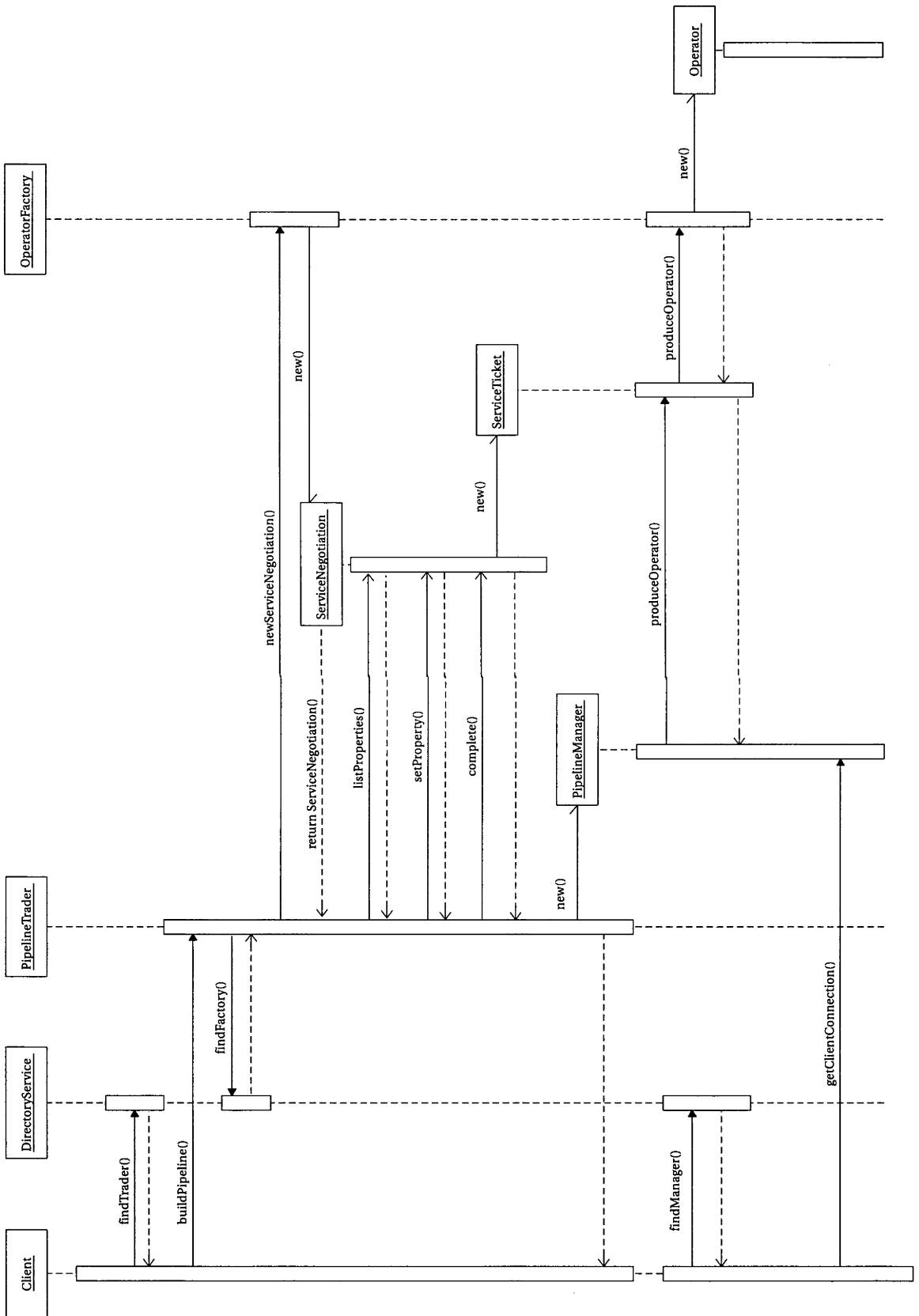
```

---

**Table 4.2:** The ServiceNegotiation interface

the operator is a public interface to the filter as it runs. For an image repository (ImageryArchive) the ticket is an agreement to make a dataset available at a certain time, while the operator (ImageAccessor) represents the runtime access mechanism into the repository.

ServiceNegotiation provides a minimal interface for accessing a resource or service *once a client knows what resource it requires*. However, if the client does not know the exact details of the resource required, a separate search interface will be required. For example, a client may know the general details of an image required from an archive, but not the specific ID of an individual image. Defining appropriate search and browsing interfaces for the myriad different operator factories is totally beyond the scope of this work. However, many of these interfaces already exist. The GIAS provides a standard interface to imagery archives; many batch queuing systems, and most computational grids, already provide interfaces to their compute facilities. Consequently, OperatorFactory provides methods to listInterfaces() and access individual interfaces (getInterface()). Once again, the names and de-



**Figure 4.2:** Example interactions in service negotiation

tails of external interfaces are not specified here, and should be included in an implementation profile.

### 4.2.3 Managing Access to Special Purpose Networks

<code>TransportFactory</code>	Manages access to special purpose networks	p. 171
<code>Transporter</code>	An operator which transports data over a network	p. 171

Special purpose, experimental and high performance networks, such as Internet-2 / Abilene [70] and Canarie [14], are valuable resources requiring careful management. They may also have constraints as to what traffic they can carry. Some connections over specialist networks are modelled as an operator in the dissemination pipeline. This allows the same resource management scheme to be used for computational and communication resources. It also provides an elegant way to overcome the routing problems that can be encountered using a specialist network.

Specialist networks may have constraints about the types of traffic they can carry; constraints that are enforced at the packet routing level. Even when a site is part of a specialist network, there may be only a small number of privileged hosts at the site which have direct access to the network. Other machines can only use the network if one of the privileged hosts is prepared to route traffic for it. Some applications that run over a specialist network have to implement application-level routers that run on a privileged host and pass appropriate data from the local network to the specialist one.

The `TransportFactory` and `Transporter`, depicted in Figure 4.1, are used to manage access to specialist networks. During service negotiation the factory decides whether an application is allowed to use the network, and makes reservations to ensure that any quality of service (QoS) guarantees can be met during service provision. The factory produces `Transporter` operators, which sit in the pipeline like any other operator and simply transport request and response messages over the specialist network. If necessary, these operators also perform the application routing required to bridge between a commodity network and the specialist network.

### 4.2.4 A Schema for Resource Discovery

To integrate the four principal actors into a computational grid requires appropriate descriptive metadata: also known as a resource schema. Table 4.3 presents the RAPID resource schema in full. There are many different formats for presenting directory schemas [60, pages 201–214], but all tend to be verbose. For clarity Table 4.3 summarises the schema in a simple declarative notation<sup>4</sup>. The schema has three major elements: factories, traders and managers relate to the construction of pipelines; sites and networks describe the topology of a distributed application; and, people and machines describe the individuals involved in a dissemination pipeline.

<sup>4</sup>Although non-standard, this notation maps easily to any standard format such as ASN.1, LDAPv3 or slapd.conf.

---

<p><b>Factory</b>  description : String  site : name of Site  category : RACE   ARCHIVE   NETWORK   FILTER  outputs : set&lt;MIMETYPE&gt;  reference : reference to OperatorFactory</p> <p><b>Trader</b>  description : String  produces : set&lt;MIMETYPE&gt;  reference : reference to PipelineTrader</p> <p><b>Pipeline</b>  description : String  reference : reference to PipelineManager</p> <p><b>Machine</b>  site : name of Site  address : set&lt;INetAddr&gt;  architecture : String  operating-system : String</p> <p><b>Person</b>  site : name of Site  firstname, surname, username : String  email : String  credentials</p>	<p><b>Site</b>  organisation : String  location : String  networks : set&lt;name of NetworkGateway&gt;  race-factory : reference to RACEFactory  manager : name of Person  users : set&lt;name of Person&gt;  hosts : set&lt;name of Machine&gt;</p> <p><b>Network</b>  description : String  use-policy : xml  links : graph&lt;NetworkLink&gt;  factories : set&lt;reference to TransportFactory &gt;</p> <p><b>NetworkLink</b>  name : String  within : name of Network  start : name of NetworkGateway  end : name of NetworkGateway  bandwidth  latency</p> <p><b>NetworkGateway</b>  site : set&lt;name of Site&gt;  link : set&lt;name of NetworkLink&gt;</p>
--	--

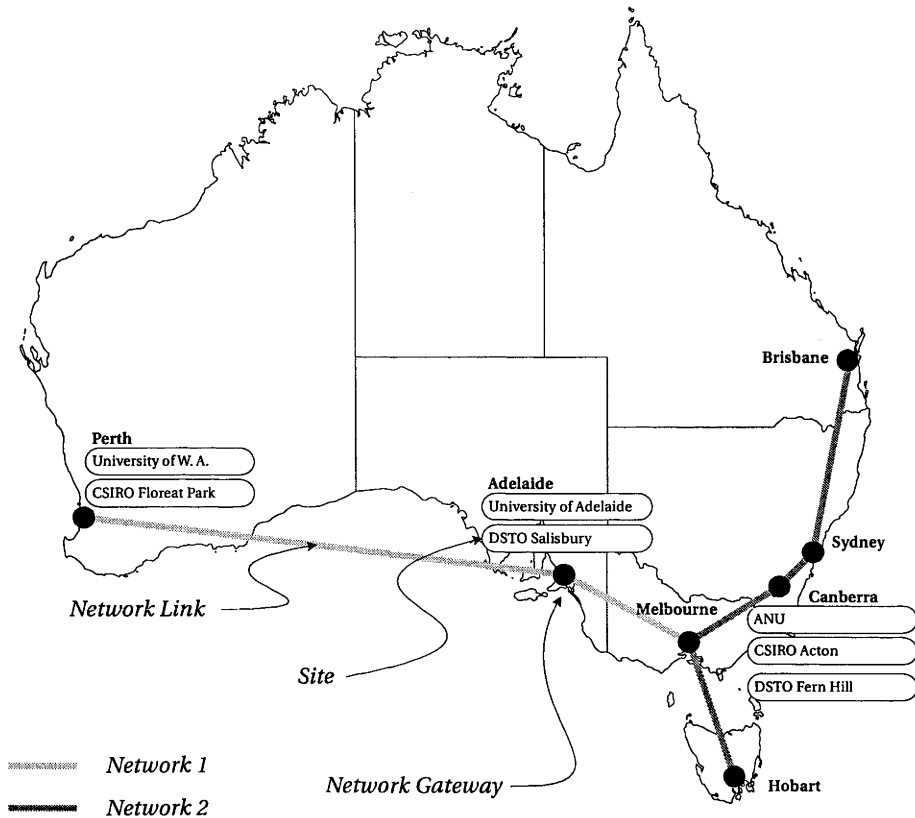
---

**Table 4.3:** The RAPID Resource Schema

**Factory, Trader, Pipeline** Three of the principal actors in the service model are represented in the resource schema. Each of these actors publishes an entry in the directory, including a «remote» reference through which it can be accessed. This allows traders (and clients) to search the directory service and retrieve a reference to a runtime object. The relationship is bi-directional. Any actor can report the path to its schema entry, with the `getDistinguishedName()` method. This returns the full, distinguished name of the actor's entry in the directory service.

Factory entries in the schema are categorised as one of four major types: RACE caches, imagery archives, computational operations (filters) or specialist networks. Each factory also lists the different types of operator it can produce, using the MIME-style enumeration of types defined in the implementation profile. Trader entries list the different types of pipeline they can produce, again relying on definitions in the implementation profile.

**Site and Network** To allow efficient routing of data within the pipeline, the schema provides a means of describing wide-area network topologies formed at the level of organisational sites. All sites (as defined in section 3.1.3) are fully connected through the commodity Internet. In addition there may be point to



**Figure 4.3:** An example illustrating the various elements of the RAPID Schema used to describe networks.

point links between sites provided by specialist, high-performance networks.

The topology of a specialist network is described through three entries in the schema: Network, NetworkGateway and NetworkLink. Examples of these entities are depicted in Figure 4.3. Network entries represent a complete specialist network, with a standard usage policy<sup>5</sup>. Access to a network occurs through a NetworkGateway, which represents a logical node in the network, such as a point of presence (POP). Site entries include a collection of machines and people located at the site, and the distinguished names of NetworkGateways to which the site has access. Multiple sites may join the network at the same gateway, with the assumption that the latency between sites at the same end of a network is negligible and routing is automatic. Connections in the network are formed from NetworkLinks, where each NetworkLink joins two gateways.

Network entries also serve to advertise a TransportFactory which can be used to move data over the network (through a Transporter). These trans-

<sup>5</sup>The encoding of network usage policies should be defined in the implementation profile.

port factories ultimately make the routing decisions. The topology information published in the directory service is designed to let traders determine whether or not to use a transport factory: the final decision of how the network is used remains with the factory.

**Machine and Person** The schema also publishes descriptions of individual machines and people. While not strictly essential, in practice it proves to be very useful to include them in the schema. Publishing machines can aid in making routing decisions and selecting computational facilities. Publishing user details is invaluable for collaboration and could also be used as the basis for authentication in a secure environment.

This simple collection of metadata is more than adequate for pipeline resource discovery. It is also in keeping with the schema design principles identified in the last chapter (see section 3.2)

It is important to draw a distinction between a resource schema and its encoding within a directory service. Although RAPID defines a resource schema, it doesn't define an encoding of the schema. In other words, it defines *what* information should be kept in a directory service, but not *how* it is kept there. The encoding is certainly a critical detail, since it affects the structure of the distinguished names of all entities. However, encoding format is highly dependent on the directory service in which the schema is maintained. It is also very sensitive to scalability requirements. For example, a schema to cover ten sites would be structured very differently to one for the entire planet. For these reasons, the encoding of the schema is left as one of the more important details in an implementation profile. Schema encoding is considered again in the vGrid case study in Chapter 7.

### 4.3 Creating the Pipeline During Service Initialisation

Negotiation is not performance sensitive and takes place over a high-level object bus, such as a CORBA ORB [105] or an equivalent grid mechanism such as Nexus [37]. Performance is critical during the provision phase, so all data is streamed along the pipeline using very efficient communication mechanisms with techniques such as parallel streaming. Constructing the pipeline and connecting operators is the task of the `PipelineManager` during the intermediate initialisation phase of the pipeline life-cycle.

A trader creates a manager with a collection of tickets, one for each operator in the pipeline, and a graph which describes how the operators should be connected to form a pipeline<sup>6</sup>. The manager starts initialising services at `startServiceTime`. To produce concrete operator objects it evaluates each of its service tickets with the

---

<sup>6</sup>As discussed in section 2.1.3, the term "pipeline" is somewhat misleading since it implies a linear sequence of processing operations. Dissemination pipelines are non-linear: forks and joins in the pipeline are supported. Hence the manager has an `operatorConnectivityGraph` where operators are the vertices of the graph, and the edges are links between the operators along which data flows.

`produceOperator()` method of `ServiceTicket`. This causes the factory that issued the ticket to construct an `Operator` object. The operator it creates is an empty shell, disconnected from any other operator. The pipeline manager connects operators in accordance with the connectivity graph, using the techniques described in section 4.3.2 below. Once an operator is connected to an appropriate set of inputs, it can be initialised by calling its `initialise()` method. Initialising the operator causes the shell to be filled out. When all operators are initialised the pipeline is ready for service provision.

### 4.3.1 The Structure of a New Operator

<code>OperatorPort</code>	Part of the service provision interface to an operator, optimised for bulk data flows	p. 174
<code>Link</code>	A reliable, duplex network connection used to send and receive data between two operator ports	p. 176
<code>Connection</code>	Used to join ports on different operators	p. 172

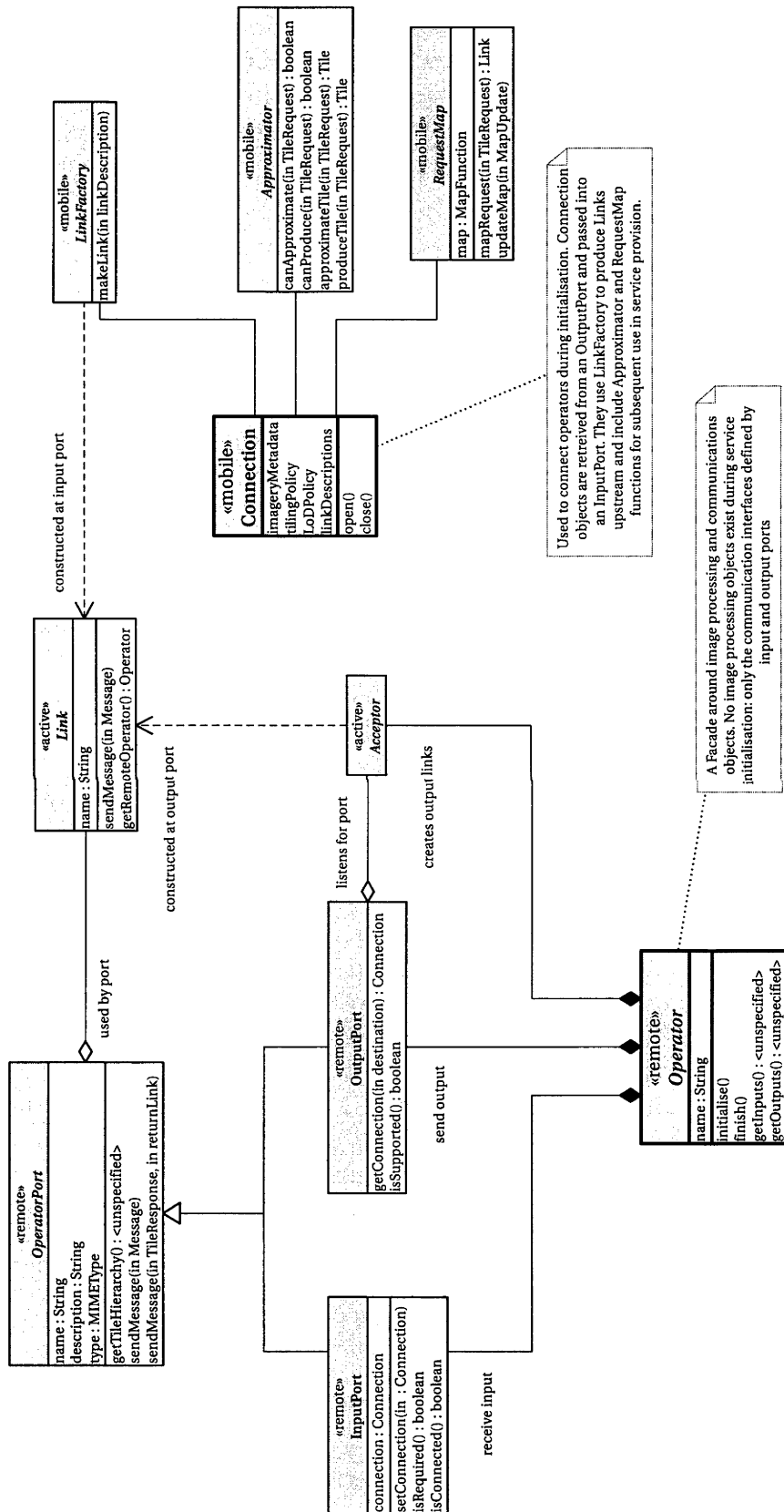
Each operator is a collection of communication and image processing objects. Not all of these objects exist when the operator is created by a factory: many are added during service initialisation. Figure 4.4 depicts the structure of an operator during service initialisation. The `Operator` class is essentially a Facade [40]<sup>7</sup>. `Operator` contains little state other than a `name` attribute, which is assigned by the creating factory and is a globally unique identifier. All other state is delegated to other classes. Operators take part in three of the four stages of the pipeline life-cycle: from service initialisation through to destruction. Consequently `Operator` is a «remote» object that may be accessed over the object bus.

To avoid using the object bus for service provision, performance sensitive communication between operators takes place through links between input and output ports. A port is a single-purpose interface to the operator, through which only one type of image data flows. Ports are used during service initialisation and must be visible to remote clients, so the base `OperatorPort` class is «remote». Each port is named, and the tuple (`Operator.name`, `OperatorPort.name`) is unique within the pipeline. Ports also have a type associated with them to describe the format of data they can handle. This is represented by a MIME-style enumeration and is provided as an aid to connecting ports of the correct type during pipeline creation. An operator receives data from an `InputPort` and sends results out through an `OutputPort`<sup>8</sup>. Each output port can send data to many different downstream operators, but each

<sup>7</sup>Note that the term “Operator” has two meanings here. At the abstract level, an operator is a component in a dissemination. This component is formed from a large number of interacting subsystems. The `Operator` class is a simple interface to the component, and acts as a Facade [40, pages 185–194] around the subsystems. To help the reader separate these two meanings, the fixed-font `Operator` is used to refer to the class. Other uses of the term refer to the general component.

<sup>8</sup>There is no requirement that all operators have at least one input and one output port. Obviously image access operators, being the sources of data, need not have any inputs. Similarly visualisation clients, as the sinks of all data, need not have any outputs.





**Figure 4.4:** Structure of a RAPID Operator during service initialisation. The Operator class and the Connection class both have particular significance during the initialisation phase.

input port can only receive data from a single upstream operator<sup>9</sup>.

Although ports define the service provision interface to an operator, data actually flows between operators along a network `Link`. The base `Link` class is an abstraction for any reliable, duplex communication mechanism. Concrete implementations of `Link` can be built from network primitives such as TCP sockets. The movement of data between operators, over links, is described in sections 4.4.1 and 4.4.2. Two operators are joined during the service initialisation phase by passing a `Connection` object from an output port of one to an input of the other. Many communication mechanisms require a passive, receiver at one end to accept new network links. Within the operator this role is performed by one or more `Acceptor` objects. The next section describes in detail how connections are formed.

### 4.3.2 Connecting Ports and Creating Links

<code>Connection</code>	Used to join ports on different operators	p. 172
<code>Link</code>	A reliable, duplex network connection used to send and receive data between two operator ports	p. 176
<code>LinkFactory</code>	An Abstract Factory [40] for links	p. 176
<code>Acceptor</code>	Receiver, used to create the upstream end of a link	p. 171

Ports are connected and links created by the `PipelineManager`, as part of the initialisation phase, with the help of `Connection` objects. To join two operators the manager retrieves a `Connection` object from a port on the upstream operator (with the `getConnection()` method of `OutputPort`) and passes it to a port of the downstream operator (with the `setConnection()` method of `InputPort`). The `Connection` class is «mobile» so instances can be moved over the object bus. Each connection contains all the necessary logic to open links to an operator, metadata for all the datasets output by an operator and descriptions of the tiling and level of detail mechanisms employed. It may also contain optional approximation functions and request map functions, both of which are described later. `Connection` and its related classes are depicted in Figure 4.4, and are summarised again in Table 4.4.

When an input port is assigned a connection, it calls the `open()` method on the `Connection` to create links upstream. This method iterates through the list of link descriptions, creating each link and registering the port as a listener to messages received on the link<sup>10</sup>.

For data to flow between two operators a `Link` object must be created on both. At the downstream operator the link is created by a `LinkFactory` object, passed in the `Connection`. A `LinkFactory` is a «mobile» object, which follows the Abstract Factory pattern, and produces instances of the `Link` class. The use of mobile link fac-

<sup>9</sup>Not all the input ports of an operator have to be connected: the `isRequired()` method of `InputPort` indicates whether an input is required or optional. However, when an optional input is omitted it may not be possible to produce all outputs. Consequently the `isSupported()` method of `OutputPort` indicates whether a particular output can be produced, based on the inputs to an operator.

<sup>10</sup>It also instructs the link about the `TileBuffer` attached to the port, as described later in section 4.4.3.

- 
1. XML descriptions of all instances shared by output (metadata)
  2. XML descriptions of tile and LoD hierarchies
  3. Factory objects to create different kinds of links (`LinkFactory`)
  4. Link descriptions (name, factory, creation param)
  5. An optional map function (`RequestMap`)
  6. An optional approximator (`Approximator`)
- 

**Table 4.4:** The contents of a `Connection` object

tories allows an operator to tailor the use of network primitives on a connection-by-connection basis. The port `getConnection()` method takes the destination as a parameter. This information may be used to select the most efficient type of link on which to build the connection. Simple operators may choose to ignore the parameter and return a connection based on TCP socket links. However, since connecting operators is not a performance critical step, an efficient operator could use a routing service or network broker to make decisions about what kind of links to use. This allows for great optimisation of the communications primitives used in the pipeline. The use of `Connection` objects is explored further in the practical work in Chapter 6.

At the upstream operator a `Link` object is created by an `Acceptor`. Many communication mechanisms require a passive listener to accept new connections. `Acceptor` objects perform this role, following the `Acceptor and Connector` [98, pages 191–239] pattern. An `Acceptor` is required for every different protocol or communications mechanism an operator uses to send data downstream. For example, if an operator can send data by TCP socket and through a shared memory buffer, then there will be one acceptor for TCP connections and a second acceptor for in-memory communications. When a request to connect is received from the underlying communications layer, the `Acceptor` creates a `Link` object and listens for appropriate signalling to determine what port the link is attached to<sup>11</sup>.

When `Link` objects have been created at both operators they can be used to send messages up and down the pipeline. When all the required inputs to an operator have been connected, and the operator has been initialised, it is ready to provide a service.

---

<sup>11</sup>This signalling comes in the form of an `OpenLink` message, described in section 4.4.2 and in Appendix A.3, page 180. The precise means by which a `OpenLink` message is sent to `Acceptor` will be protocol dependent. For TCP sockets the message should be sent along the new socket. This detail is encapsulated within the `LinkFactory` sent to the client, and so is not significant.

## 4.4 Provision of Service with a Dissemination Pipeline

TileBuffer	Stores the input and output of an operator	p. 177
Approximator	Approximates some tiles not in the buffer	p. 172
RequestHandler	Handles all messages sent along the pipeline	p. 176
Transformation	Performs image processing of tiles	p. 178

The service provision phase of the pipeline life-cycle begins only when all operators are initialised. The structure of the operator changes during service provision, and is depicted in Figure 4.5. Gone are the acceptor and connection objects used to join operators together, and the factories used to create new links<sup>12</sup>. All communications now takes place over the service provision interface of ports and links. The movement of data up and down the pipeline is described in section 4.4.1. A very limited set of signalling is required between operators. This signalling is known as the RAPID operator protocol (or RAPPORT), and is described in section 4.4.2.

To assist in the movement of bulk data (tiles) through the operator, every input and output port uses a `TileBuffer`. An `Approximator` function may be associated with each buffer, to produce a temporary approximation of a tile when it is not cached. This is in keeping with the use of approximations described in section 3.1.7 of the last chapter. The use of buffers and approximation are described in section 4.4.3.

Processing requests and results from other operators is the job of a `RequestHandler`. Handlers do not perform the actual image processing tasks, but simply respond to messages received from ports. Image processing is performed by a `Transformation` function. Request handling and transformation are described in detail in section 4.4.4. The structure of an operator is designed to make it easy to support various forms of parallelism, including parallel processing within an operator and parallel streaming between operators. These techniques are considered in section 4.4.5.

### 4.4.1 Basic Network Communications

Link	A reliable, duplex network connection	p. 176
OperatorPort	Logical interface to the operator: a collection of links to multiple operators or nodes	p. 174
RequestHandler	Handles all messages sent along the pipeline	p. 176
RequestMap	Used in downstream gathering to select between links	p. 177

The movement of messages between operators is managed by three classes: `Link`, `OperatorPort` and `RequestHandler`. `Link` is an abstract class which encapsulates a duplex network connection that can send and receive RAPPORT messages. This closely matches the behaviour of a socket, but could easily be built on top of many

<sup>12</sup>There is no strict requirement that operators can only accept connections during service initialisation. An operator may choose to preserve its `Acceptor` and `Connection` objects and so allow new operators to join at any time. This is, in fact, essential to the operation of the RACE cached since clients do not join the pipeline until after it is fully initialised. See section 4.6 for more details.

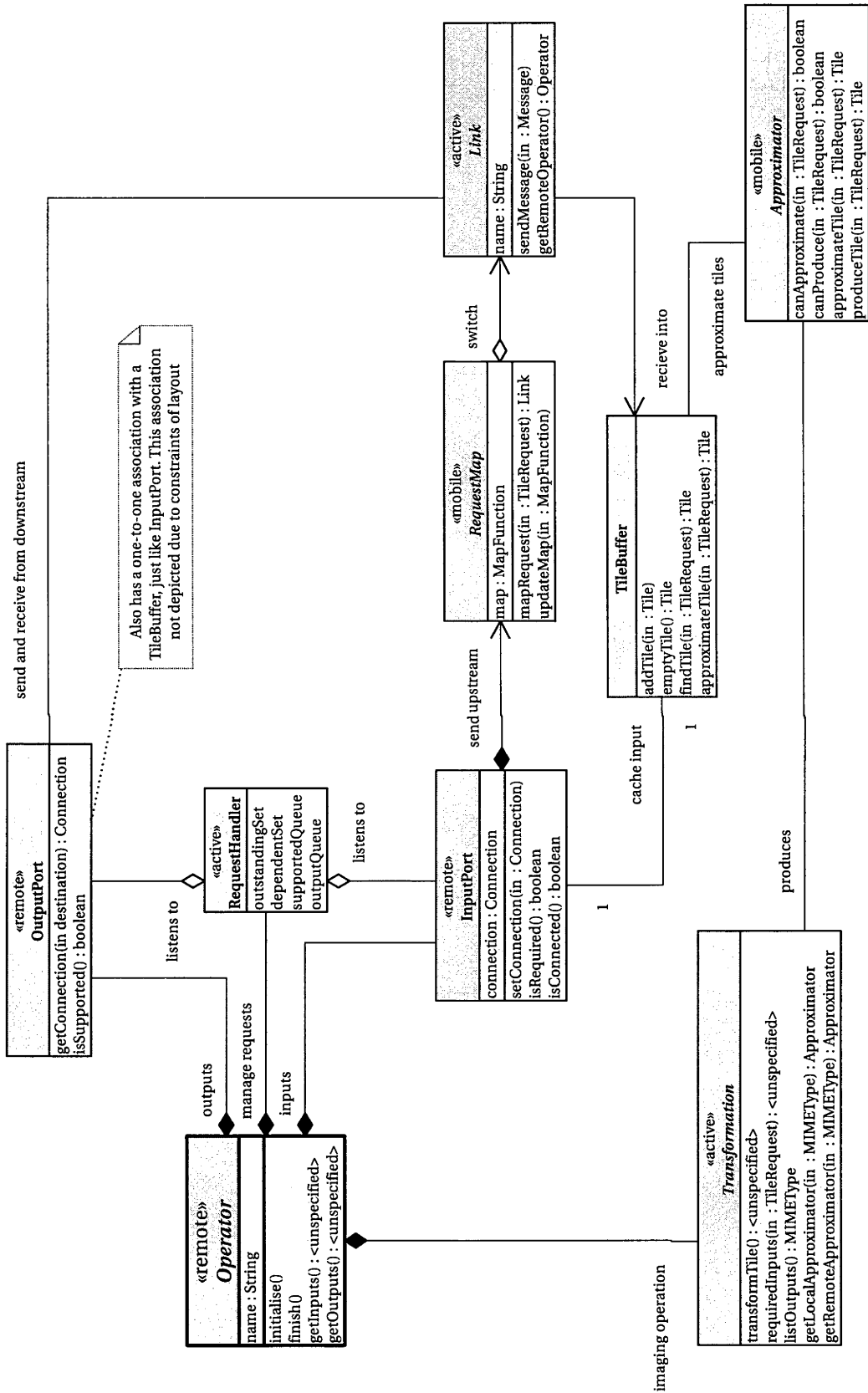


Figure 4.5: The RAPID Operator class and related classes used during service provision. Not all associations between classes are depicted in this diagram.

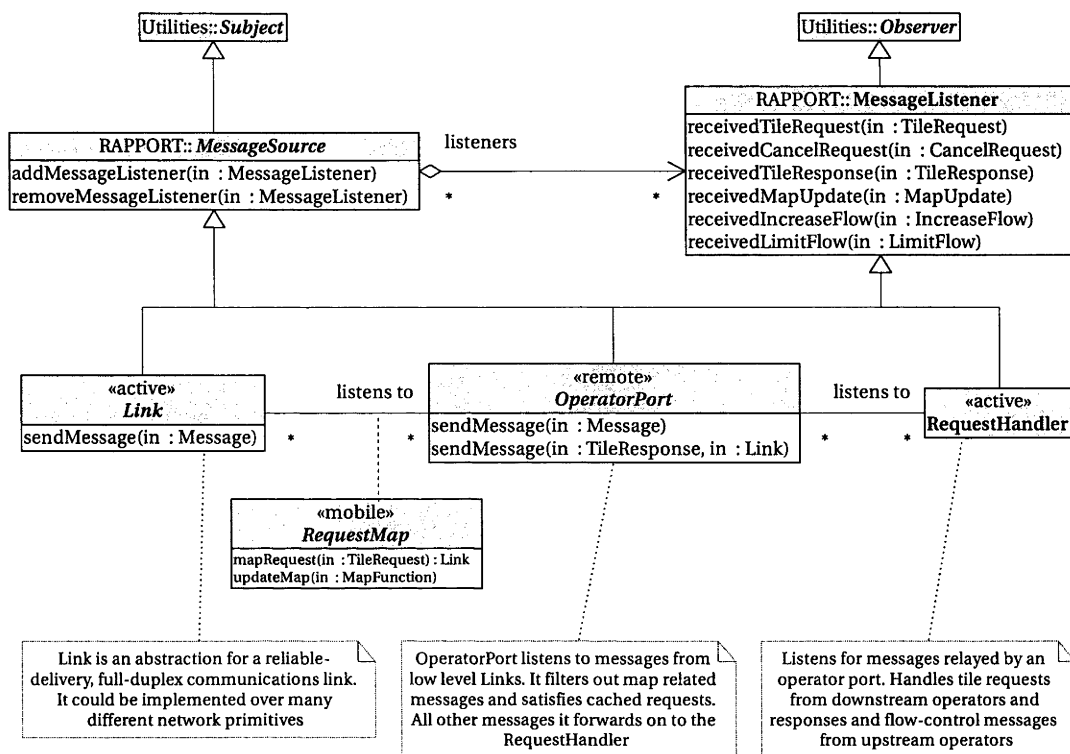


Figure 4.6: Use of the Observer Pattern in basic communications objects

other communication mechanisms. Each operator can have a number of input and output ports: each port will have one or more links to ports on other operators. Ports listen for messages from links and can respond in limited ways, but the major logic for request processing resides in the RequestHandler class. Figure 4.6 demonstrates the static relationships between these classes, based on the Observer [40, pages 293–303] and Reactor [21, pages 529–545] patterns.

The handling of newly received RAPPORT messages follows the Observer pattern. Link provides a listenable *Subject* interface for objects interested in incoming messages. Every time a message is received the link invokes a callback method in each listener passing it the new message. OperatorPort listens to the messages received from the link. It can handle some messages directly<sup>13</sup>, but forwards most on to the operator’s RequestHandler. Handlers do not listen directly to the network links, but instead listen to the ports. So, a short chain of message handlers is formed with messages starting at a Link and being relayed through an OperatorPort to a RequestHandler.

The interaction between threads in these three objects is notable. Link is an «active» object because many network primitives are synchronous in nature and a

<sup>13</sup>The MapUpdate and some TileRequest messages, described in section 4.4.2.

separate thread may be required for asynchronous messaging. When a new message is received from the network, the thread associated with the link calls the appropriate receiver method in each `MessageListener` registered with it; typically just the port. Listeners should not perform major computation within this callback. The filtering undertaken by `OperatorPort` is cheap and can be performed within the callback. However, handling a client request is potentially expensive so `RequestHandler` is an «active» object that runs with its own thread of control. The receiver callbacks within `RequestHandler` simply move each new message in a prioritised queue, and process them in priority order. Consequently the lifetime of `Message` objects created by a `Link` must be managed with care<sup>14</sup>.

Sending a message involves a similar chain of calls to that to receive a message. `Link` has a `sendMessage()` method for outgoing messages, but most messages stem from a `RequestHandler` which operates at the level of ports rather than links. When the handler sends a message it calls the `sendMessage()` method of `OutputPort`. In most cases the handler will leave it to the port to decide which links the message should be sent on. The choice of link is usually simple, except for downstream gathering forms of parallel streaming. This style of streaming allows a port to have a direct link to each node within a parallel operator upstream. The problem arises when sending a message upstream, since the port must choose which link to send the message on. The port uses a `RequestMap` function, provided during service initialisation in the `Connection` object, to map tile requests to physical links. Upstream scattering can also affect the choice of link. Both forms of parallel streaming are discussed in more detail in section 4.4.5.

#### 4.4.2 RAPPOR: The RAPID Operator Protocol

---

Message	The base of the RAPPOR message hierarchy	p. 181
---------	--	--------

---

A limited set of signalling is required between operators in a pipeline. Request messages flow up the pipeline: responses flow down the pipeline. Some additional signalling is required to support flow control within the pipeline. Downstream gathering forms of parallel streaming use a request mapping function, which may also require signalling to stay up-to-date. These few requirements lead to a simple protocol for sharing data between operators, known as RAPPOR.

---

<sup>14</sup>Asynchronous processing of callbacks raises the question of how and when message objects are deleted. For runtime environments that support automatic garbage collection this problem is trivial. For those that do not, there are two obvious approaches. First, a simple reference counting system could be added to the basic `Message` object. Listeners that wish to refer to a message outside the receive callback should increment the message's reference count within the callback, and decrement it when they have finally finished with the message. Reference counting works well since there are never cyclic dependencies between messages. An alternative approach is to require that listeners make a copy of any message they wish to use outside the callback. All RAPPOR messages, with the exception of `TileResponse`, are very small and cheap to copy. `TileResponse` messages deliver imagery along the pipeline and account for the greatest volume of traffic. Copying these messages would be a major performance problem, and we will see shortly how this is avoided. No single message deletion policy is mandated in RAPID: the above approaches are equally valid when automatic garbage collection is not available.

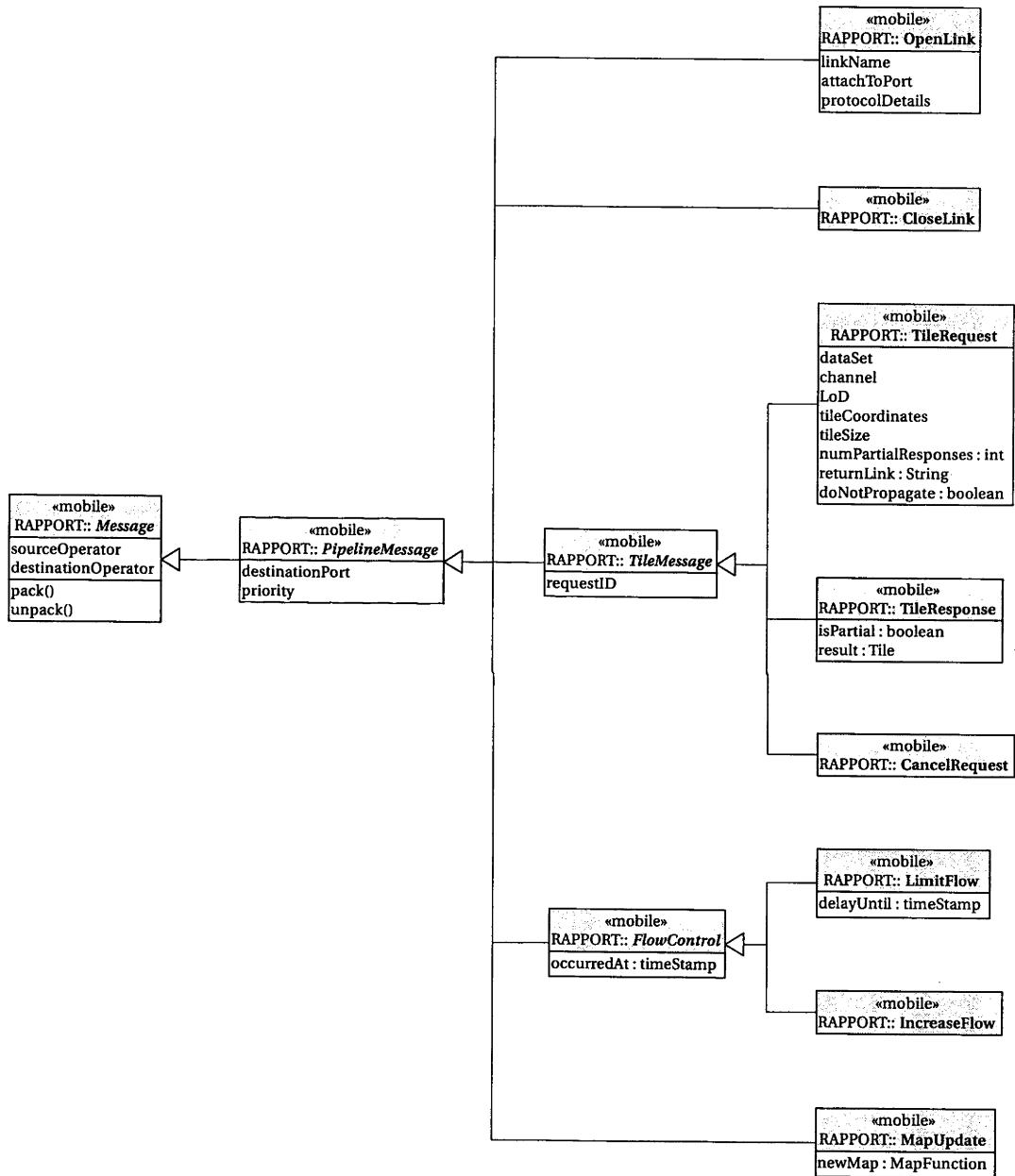


Figure 4.7: The RAPPORT Message hierarchy



Figure 4.7 shows the various message objects in the RAPPORT protocol<sup>15</sup>. All messages are descendants of an abstract class `Message`, which is a mobile object. Messages sent along the pipeline (as distinct from collaborative messages) are further descended from `PipelineMessage`. Between them these classes have four attributes: the string name of the operator that sent the message<sup>16</sup>, the destination operator and port, and a priority value to expedite delivery of critical messages<sup>17</sup>. If no destination is specified the message is intended for all operators. In Figure 4.7 `Message` is also shown as having explicit `pack` and `unpack` methods but these are only required for runtime systems without automatic object serialisation. No precise packet-level structure is defined for RAPPORT messages, since it will vary from one transport mechanism to another. Serialisation mechanisms and associated packet structure are details that should be provided in an implementation profile. Profile issues aside, RAPPORT should be trivial to implement over any reliable, in-order communication mechanism<sup>18</sup>.

There are three tile-related messages defined within RAPPORT: to make requests, cancel requests and respond to requests. All three message types are descended from a common abstract ancestor, `TileMessage`, which contains a `requestID` attribute. Every tile request is assigned an ID at its source operator. This ID is unique at the source operator and consequently the tuple (`sourceOperator`, `requestID`) uniquely identifies a tile request within the pipeline. `TileRequest` messages are sent upstream when an operator requires a tile: `TileResponse` messages flow down the pipeline and contain the data to satisfy a request. Most of the attributes of `TileRequest` come straight from Table 3.2 in the previous chapter. It also includes a `numPartialResponse` attribute, to support the use of approximations described in section 3.1.7. This specifies the maximum number of partial responses (approximations) that can be sent in lieu of the correct result. The upstream operator does not have to send any partial responses, but if it does choose to send approximations this attribute limits the number sent. `TileResponse` has a corresponding `isPartial` attribute to identify whether a response is final or is a temporary approximation. `TileRequest` also has an attribute to describe the link on which return messages should be sent (`returnLink`). This is used to support upstream scattering forms of parallel streaming, as described in section 4.4.5.3. Finally the `CancelRequest` message is provided to support eager pipeline requests, as described in section 3.1.5. This message requires no additional attributes beyond the (`sourceOperator`, `requestID`) tuple defined in its ancestors which unambiguously identifies the request to be cancelled.

RAPPORT includes two simple messages used during creation and destruction of network links. An `OpenLink` message is sent from a downstream operator to an `Acceptor` of an upstream operator when a new link is opened. The major attribute of `OpenLink` is a unique name or identifier for the new link. Additional, protocol spe-

---

<sup>15</sup>These are also summarised in Appendix A.3 on page 178.

<sup>16</sup>Each operator has a `name` attribute that is unique within the pipeline.

<sup>17</sup>The range of priority values is not specified here: it is an implementation detail and should be part of an implementation profile.

<sup>18</sup>the need for in-order delivery could also be relaxed quite trivially, but is not an unreasonable.

cific data may also be provided. The `CloseLink` message class is the obvious partner to `OpenLink`, and is used to perform graceful service shutdown. Protocol-specific contents are also allowed, and should be specified in an implementation profile.

A very simple model of flow control is defined for the dissemination pipeline to help throughput. Two flow control related messages are defined as descendants from an abstract `FlowControl` class. If an operator is not running at peak capacity it may choose to send an `IncreaseFlow` message to one or more downstream operators connected to its output ports. This message is a general indicator only, and downstream operators can respond to it in whatever way they see fit. If an operator is running at peak capacity it may simply ignore this message. If not, it may send the message on down the pipeline. In this way the `IncreaseFlow` messages will eventually reach the pipeline schedulers running at each site cache, who may choose to respond to them by increasing the number of speculative requests made of the pipeline. The `LimitFlow` message is the natural opposite of `IncreaseFlow` and provides a mechanism for constraining the transmission of requests up the pipeline. Limit messages are usually sent downstream with a very high priority so they reach the pipeline scheduling mechanisms as fast as possible.

Finally, RAPPOR defines a `MapUpdate` class of message to support downstream gathering in the dissemination pipeline. This style of parallel streaming uses a mapping function to choose the correct link when sending requests upstream. The mapping function is provided during service initialisation, but since this map function may change over time the `MapUpdate` message provides a way to update functions in downstream operators. The contents of the message are deliberately not specified. Mapping functions may be implemented many different ways, and should be specified in any implementation profile.

This small set of messages is all that is needed to operate the dissemination pipeline. RAPPOR includes an additional set of messages to support collaborative data sharing, which are described in section 4.5.5.

### 4.4.3 Port Buffers and Approximating Results

<code>TileBuffer</code>	Stores the input and output of an operator	p. 177
<code>Approximator</code>	Approximates some tiles not in the buffer	p. 172

The asynchronous behaviour of the pipeline requires that each operator be able to buffer a number of tiles. Buffering output is useful when multiple downstream operators are connected to an output port, as described in section 3.1.1. Input buffering is also required when receiving `TileResponse` messages to avoid copying overheads. So a `TileBuffer` is associated with each `OperatorPort`<sup>19</sup>. The size of buffers is obviously operator and port dependent: the larger the buffers the better<sup>20</sup>. Each link has a reference to a buffer on an input port, and receives response messages directly into the buffer. Input buffering can also be very important when the tiling and level

<sup>19</sup>Note that this association is *not* clearly depicted in Figure 4.5, due to constraints of layout.

<sup>20</sup>This may be a point of negotiation between Trader and Factory during service negotiation.

of detail mechanisms used by one operator do not match those of another. For example, if an operator outputs tiles that are smaller than it receives as input, each input tile may be used to produce multiple output tiles. Hence, it makes sense to cache each input tile as long as possible.

Buffers can also use an `Approximator` function to provide a temporary result when a tile is not in the buffer. Approximations should be very cheap to produce, since they are usually needed within a link message callback. For output ports the approximation functions are set when the port is created. A useful approximation on an output port is to return a low-resolution version of a tile. To support approximations on input ports, the base `Approximator` class is «mobile» and instances can be included in the connection object. This allows an operator to control how approximations are used downstream: a powerful capability since the `Approximator` can be used to filter requests before they are sent up to an operator.

#### 4.4.4 Handling and Processing Tile Requests

<code>RequestHandler</code>	Handles all messages sent along the pipeline	p. 176
<code>Transformation</code>	Performs image processing of tiles	p. 178
<code>OutstandingSet</code>	Contains requests received from downstream	p. 175
<code>DependentSet</code>	Contains requests sent upstream	p. 173
<code>SupportedQueue</code>	Requests which are ready for transformation	p. 177
<code>OutputQueue</code>	Requests after transformation, waiting to be sent	p. 175

Handling and scheduling requests is performed within an operator by one or more `RequestHandler` objects. These listen to output ports for requests for new tiles to be sent downstream, and to input ports for responses to requests sent upstream. There are four main stages in handling a tile request: fetching the inputs to the request; waiting for a transformation to perform the processing; processing the request; and waiting for the response to be sent downstream. Supporting approximations adds some subtleties to request handling, as demonstrated in Figure 4.8. The `RequestHandler` uses four collections to manage requests, and at any moment the state of a request is reflected by the collections in which it is present.

**OutstandingSet (p. 175)** Requests from downstream operators that have not had a final response sent to them are held in an `OutstandingSet`. Each entry in this set contains the parameters of the requested tile, along with tuples (`requestID`, `sourceOperator`, `returnLink`) for each downstream operator that has made a request. These tuples are used to select the links on which to send a response, along with the ID to send with each response. Entries also contain the `requestID` of dependent requests made to upstream operators. The `OutstandingSet` needs to support three different lookup methods. To check for simultaneous requests by downstream operators it must support fast access based on the parameters of a tile request. To aid processing data received from upstream it should also support fast access based on the `requestID` of dependent requests. Finally, it should support reasonably fast access based on the tuple (`sourceOperator`, `requestID`) for the purposes of request cancellation.

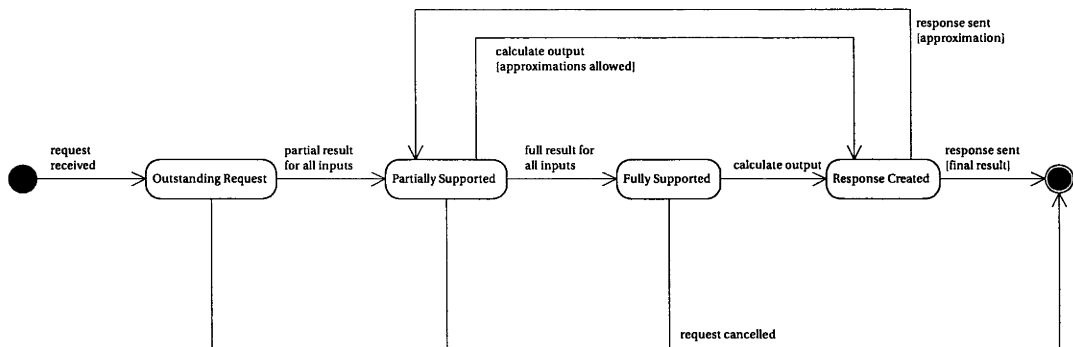


Figure 4.8: The different states in handling a tile request

**DependentSet (p. 173)** The set of requests that have been made to upstream operators is kept in a `DependentSet`. Each entry contains the parameters of the tile request, and a `requestID` which is unique within this operator. The `DependentSet` should support fast access to entries based on the `requestID`. When a response is received the entry in the dependent set is updated to include the buffer in which the tile is kept. An entry is kept in the dependent set until the original request is removed from the `OutstandingSet`.

**SupportedQueue (p. 177)** Contains all the downstream requests that are partially or fully supported, and so are ready for transformation. This is the work queue for `Transformation` threads. They remove entries from the front of this queue, perform the associated processing, place the results in the buffers of all appropriate output ports, and register the completed request in a queue of completed results. The `SupportedQueue` is prioritised based on the original request priorities, so that demand-driven requests are processed before speculative requests.

**OutputQueue (p. 175)** Contains requests that have been processed and are ready to be sent back downstream. Again this queue is ordered by request priority.

A `Transformation` object performs the actual processing of tiles, and encapsulates the logic of an image processing operation. Image processing can be computationally expensive so `Transformation` is an «active» object running in a separate thread. The `SupportedQueue` contains work items for the transformation thread. This queue contains requests from downstream operators for which all the required inputs are available. To identify what inputs are required for a tile, the `RequestHandler` uses the `requiredInputs()` method of `Transformation` to make dependent requests to upstream operators. The actual transformation is performed by the `transformTile()` method. It removes a single request from the `SupportedQueue`, performs the transformation, places outputs in the tile buffers of all relevant output ports and moves the completed request to the `OutputQueue`.

Transformations also serve as factories for the `Approximator` functions used on

tile buffers. Local output ports may use a different approximator to downstream input ports, so there are separate methods for retrieving `Approximator` objects for local use, and for inclusion in a `Connection`<sup>21</sup>. A transformation function does not have to support either type of approximation function, but can choose to support one or both as appropriate.

The procedure for handling tile requests starts at the `OutputPort`. When a request is first received by an `OutputPort` it checks the `TileBuffer` to see if the result has already been computed. If so, an appropriate `TileResponse` is sent back downstream. If not, the port may send an approximation if one can be produced and is acceptable to the downstream client<sup>22</sup>. At this point the request is passed on to the `RequestHandler` which is responsible for producing the correct result. From here on the handler follows the four stages of request processing described above. The handler starts by using the transformation function to determine what dependent requests must be made to upstream operators<sup>23</sup>. The original request is placed in the `OutstandingSet` and the dependent requests are placed in the `DependentSet` and sent upstream with the `sendMessage()` method of the `InputPort`. Before sending each request the input port checks its `TileBuffer` to see if the request has been cached locally. If so, it creates an appropriate `TileResponse` message and calls the `receivedTileResponse()` callback in the `RequestHandler`. If not, the `InputPort` may produce an approximation, which is passed back through the same callback<sup>24</sup>. Next the port sends the message upstream along the appropriate `Link`, using a `RequestMap` to select the link if more than one link has been opened upstream. At this point processing of the original request halts until responses are received for each dependent request.

Tile response messages are also processed by the `RequestHandler`. When a `Link` receives a response message it places the payload of the message directly into the port `TileBuffer`<sup>25</sup>, and then notifies all listeners<sup>26</sup>. Normally the `InputPort` will listen to the link, and pass the message along to the `RequestHandler`<sup>27</sup>. When a handler receives a response message it uses the `requestID` to find the entry in the `DependentSet` corresponding to the dependent request. It updates this entry to include details of the response that was received, then finds the original entry in the `OutstandingSet`. If a response has been received for all dependent requests, the original request is copied into the `SupportedQueue` in priority order.

---

<sup>21</sup>`getLocalApproximator()` and `getRemoteApproximator()` respectively. Both methods take as a parameter the type of output to be approximated.

<sup>22</sup>Being careful to decrement the `numPartialResults` counter in the request message.

<sup>23</sup>Through the `requiredInputs()` method of `Transformation`.

<sup>24</sup>As before, the port must decrement the `numPartialResults` counter within the request message to account for the fact that an approximation has been produced.

<sup>25</sup>This is done with the `emptyTile()` method.

<sup>26</sup>Listeners are notified through the `receivedTileResponse()` callback.

<sup>27</sup>The reason for decoupling handlers from links is to allow the local port to return approximations without a handler requiring any additional logic. This applies to both input and output ports. An additional reason for decoupling inputs is to allow the port to filter `MapUpdate` messages. Note the implication of using approximations to provide partial support for a request, identified in section 3.1.7.

---

Work items are fed to the Transformation threads through the SupportedQueue. When a thread is ready for more work it removes the front item from the SupportedQueue, and processes it. When processing is complete the request is copied into the OutputQueue and the transformation moves on to a new work item. The results of the transformation are placed the TileBuffer of all appropriate output ports.

The final stage of request processing involves sending a response to all downstream operators that made a request for the tile. The RequestHandler reads completed requests from the OutputQueue and sends response messages through the relevant output ports. When the handler removes a complete request from the queue it finds the corresponding entry in the OutstandingSet to determine what downstream connections to send responses to. It sends out TileResponse messages and then removes all relevant entries from the OutstandingSet and the DependentSet.

Some subtleties to request processing may not have been obvious from the preceding description. Concurrent access introduces considerable complexity into the pipeline, and simultaneous requests for the same tile must be handled with care. In some circumstances it is not obvious when two requests correspond to the same tile<sup>28</sup>. It is worth noting that these issues only occur when there is a fork in the pipeline: i.e. when two or more downstream operators are connected to the same upstream operator. This is most likely to occur at the very end of the pipeline. Certainly it will occur when multiple clients connect to a shared site cache (RACE). It will also occur when multiple site caches connect to a shared final processing stage of a pipeline. In both cases the behaviour of the RequestHandler serialises the request stream and coalesces duplicate requests. Much of the complexity in request handling can be removed for some operators. In section 4.5.3 we will see how a simplified handling procedure can be used in a RACE cache.

#### 4.4.5 Supporting Parallelism in the Pipeline

Much of the complexity in the operator model is due to the need to support parallelism in the pipeline. The model automatically supports intra-operator parallelism (parallelism of computation) through use of the Master-Slave [21, pages 133–142] pattern. Parallel streaming is also supported to optimise the path of bulk data transfers (TileResponse messages). Connection objects also make it possible to support upstream-scattering and downstream-gathering forms of streaming quite transparently.

---

<sup>28</sup>For example, imagine an operator with one input port  $I$  and two output ports ( $P$  and  $Q$ ) and a different downstream operator connected to each port. A request  $p$  is received on  $P$  which requires that a dependent request  $i$  be sent upstream. Before a response arrives for  $i$ , a second request  $q$  is received on  $Q$ . If  $q$  requires the same inputs as  $p$ , it will produce the same dependent request  $i$ . So before a dependent request is made for  $q$  the DependentSet must be checked to see an equivalent request  $i$  is already pending. Furthermore, if requests  $p$  and  $q$  require the same processing, they can be satisfied in a single transformation. Consequently they should appear in the same entry in the OutstandingSet. This highlights the care that must be taken in maintaining the various request sets, to avoid making duplicate requests.

#### 4.4.5.1 Intra-operator parallelism

Intra-operator parallelism allows operators to run on a high performance computing platform, such as a shared memory symmetric multiprocessor (SMP) or a multi-computer. The basic approach is task-parallelism following the Master-Slave [21] design pattern, where a single `RequestHandler` acts as master and apportions work to multiple slave `Transformation` threads through the `SupportedQueue`. The crucial shared data structures in this model are the `SupportedQueue`, the `OutputQueue` and the `TileBuffer` objects. Implementations of these structures will vary depending on whether a shared memory space is available.

On shared memory machines the challenge is to minimise the synchronisation required between threads which access the shared structures. In particular the `TileBuffer` objects could become a bottleneck since they must also be shared with `Link` threads. The granularity of thread synchronisation depends on the cost of transforming tiles and the number of concurrent threads. Fine grain synchronisation, with the greatest overheads, will be required for cheap transformations and large numbers of threads. A solution to this problem is to increase the size of tiles processed and so make each transformation more expensive. If the average request size is smaller than that required to minimise synchronisation, an `Approximator` could be attached to each output `TileBuffer` to subset tiles when requested<sup>29</sup>.

On multi-computers and clusters, with many independent compute nodes and no shared memory, the challenge is to minimise communication between nodes. One node acts as master, running the `RequestHandler`, maintaining the shared structures and running other operator elements such as the ports and links. The remaining nodes simply run `Transformation` tasks and communicate purely with the master. Communication between nodes takes place over the multicomputer's internal communications network, using an implementation dependent protocol<sup>30</sup>. Operator-internal communications are fairly obvious, and have to allow transformation threads to: remove elements from the `SupportedQueue`; add elements to the `OutputQueue`; read tiles from the input buffers and; place their results in the output buffers.

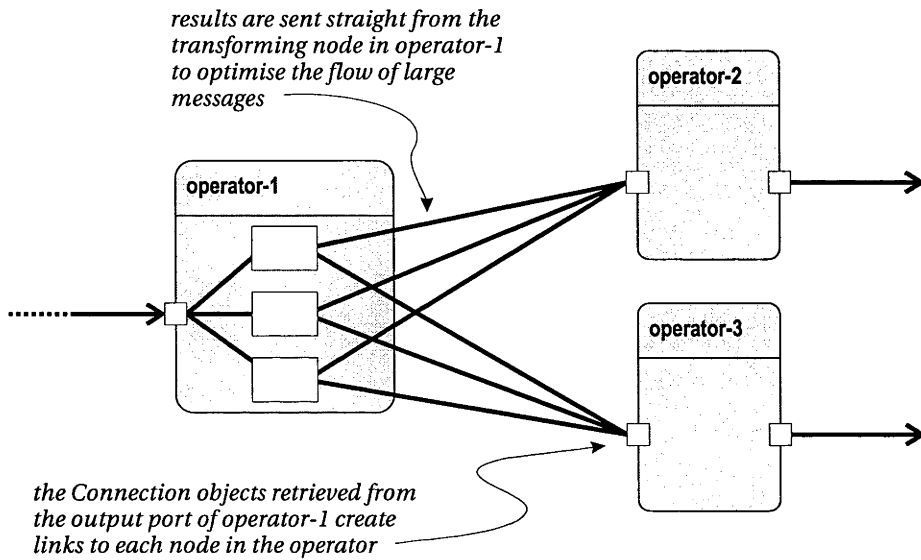
#### 4.4.5.2 Parallel Streaming Through Downstream Gathering

Downstream gathering optimises the movement of bulk data (`TileResponse` messages) out of a parallel operator. Operators downstream of the parallel operator open a link to each transformation node and the output of a transformation is sent directly to the requestor. This minimises the response time for a request once it has been processed. Figure 4.9 depicts a simple example of downstream gathering.

---

<sup>29</sup>This amounts to a form of `Request by Proximity`: a valuable heuristic which is explored further in Chapter 5.

<sup>30</sup>For multi-computers with very low-latency internal communications, an interesting alternative is to store the shared structures in a software Distributed Shared Memory [111] such as `Munin` [16] or `Treadmarks` [3]. This potentially increases the latency when accessing a `TileBuffer`, and so may not



**Figure 4.9:** Downstream gathering

The use of downstream gathering is transparent at the input to an operator. The `Connection` object used to connect two operators contains all the logic required to perform downstream gathering. In the example depicted in Figure 4.9 the `Connection` objects passed to `operator-2` and `operator-3` describes the three `Link` objects that must be created to communicate with the nodes in `operator-1`. They also contain a `RequestMap` function used to choose between links when sending requests upstream. The configuration of a connection is performed at run-time, so developers of operators do not have to be concerned with whether the inputs are serialised or use downstream gathering.

Operators that want to output results through downstream gathering do, however, require some changes. One major change is to discard the `OutputQueue`, and make `Transformation` tasks responsible for sending `TileResponse` messages to all downstream operators that have made a request. There are numerous implications of this change:

- the send list, maintained in the `OutstandingSet` must be sent to the transformation node
- entries must be cleaned up from the `OutstandingSet` and the `DependentSet`
- elements of the output port have to be replicated on every transformation node

Request handling and tile buffering are also harder to implement when downstream gathering is used. The very simplest approach is to keep the handler and tile buffers

---

be appropriate for loosely-coupled cluster computers.



on the master node. If the `RequestMap` sent to downstream clients ensures that every `TileRequest` is sent to the master node (running the handler) transformation nodes will never receive request messages. Interactions between the `Transformation` task and the `RequestHandler` are similar to those of Master-Slave parallelism, and take place via the `SupportedQueue`. An additional interaction is required after a transformation has completed, so that the transformation can determine the links on which to send results, and can instruct the handler to clean up the request sets. One limitation: as before the transformation node needs to be able to access and add to every shared `TileBuffer` on the master node.

A more sophisticated approach, is to replicate portions of the output port on each transformation node. Having a `TileBuffer` on all nodes provides the potential to buffer significantly more output from an operator<sup>31</sup>. Running an `Approximator` on every transformation may also be valuable since a more expensive approximation function could be used without affecting the throughput of the operator. These advantages come at the cost of more complex request handling. Since clients can send a request to any transformation node, each node uses a `Proxy` [40] request handler to listen to requests and forward them on to a master handler. The use of proxy handlers is considered in detail in section 4.5.

#### 4.4.5.3 Parallel Streaming Through Upstream Scattering

Support for upstream scattering of inputs is quite simple when compared to downstream gathering of outputs. Upstream scattering optimises the movement of bulk data (`TileResponse` messages) into a parallel operator, as each response is sent directly to the node that requires it. The most efficient way to move data through a parallel operator is to use upstream scattering of its inputs, and downstream gathering of its inputs. The RAPID site cache (RACE) is an example of such an operator.

Upstream scattering affects the inputs to an operator, but does not affect the output of upstream operators. In essence the `InputPort`, `TileBuffer` and `Link` objects are replicated at every node in the operator. Requests are still sent from the master node, but with some refinements to the standard approach. The `RequestHandler` specifies the name of the return link on which results should be sent<sup>32</sup>: upstream operators use this name to select the return link to send responses on<sup>33</sup>. This ensures that the `TileResponse` message is sent to, and buffered on, the node that requires it. It does, however, require that the handler know which node will process the request at the time it makes dependent requests. It may also require that buffers on input ports are able to snoop on another.

---

<sup>31</sup>This is the basis for parallel caching described in section 3.1.3 and the RACE component presented in section 4.5.

<sup>32</sup>This is specified via the `TileRequest.returnLink` attribute

<sup>33</sup>Link selection uses a specialised form of the `sendMessage()` method of `OutputPort`, alluded to earlier in section 4.4.1.

## 4.5 RACE: The RAPID Cache

The centrepiece of RAPID is a site cache, known as the RACE. The purpose of the RACE is to sit between visualisation clients and the dissemination pipeline, to decouple the execution of one from the other. A RACE runs at every end-user site and provides visualisation clients with very low latency access to the tiles of data, to meet the fundamental requirement for *responsiveness*. It also schedules and optimises the flow of requests to the pipeline to meet the fundamental requirement for *throughput*. Finally, it acts as a router for data sharing between clients, to meet the fundamental requirement for *collaborative data sharing*.

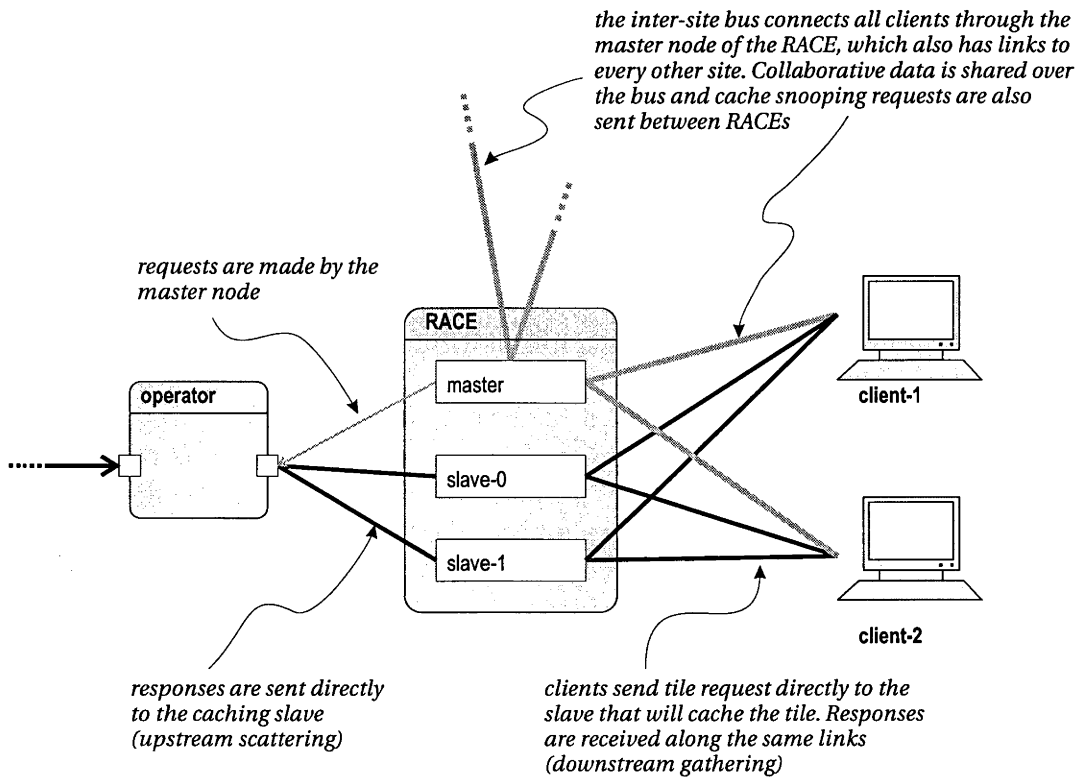
### 4.5.1 Structure of the RACE

RACE	A parallel site cache operator	p. 184
Slave	A node within a RACE which caches tiles	p. 186

The RACE acts as an operator in the dissemination pipeline, just like any other, but its behaviour is highly specialised. As an operator the RACE has these defining characteristics:

- RACE is a parallel caching operator, which runs on all free machines at a site and buffers very large amounts of data. It loosely follows the Master-Slave pattern with many Slave nodes to cache tiles, and a single Master node to coordinate pipeline requests.
- No Transformation function is run on the Slave nodes: they simply cache data in a port TileBuffer. An Approximator may be used to provide partial results when a cache miss occurs. The most common approximation is simply to return a low-resolution copy of the missing tile.
- The input and output ports on each slave use a shared TileBuffer. Tiles are received directly into the buffers and sent directly from them.
- There is one input port and one output port for every data type presented to client applications. This allows for many different types of data to be produced along independent pipelines, then stored in a single cache.
- RACE supports downstream gathering (parallel streaming) at visualisation clients to minimise response time for cached tiles. Each client has direct link to every node in the cache and a RequestMap which sends a request directly to the node caching a tile.
- RACE supports upstream scattering so that bulk data is transferred from the last processing operator in the pipeline directly to the caching node in the RACE. This makes most efficient use of the bandwidth of the local area network at each site.

The RACE performs two additional functions outside the scope of the operator model:



**Figure 4.10:** General structure of the RACE

1. it runs a speculative request service which uses one or more of the pipeline scheduling policies defined in section 3.1.2; and
2. it maintains an inter-site sharing bus to support cache snooping and routing collaborative messages between sites, as described in sections 3.1.4 and 3.1.5.

The general structure of the RACE is illustrated in Figure 4.10, while the design details are presented in Figure 4.11. A RACE operator acts as the master node for the cache. It runs a special request handler known as a `RACEHandler`, a speculative request Scheduler and handles collaborative data sharing via the inter-site sharing bus. The master does not have a `Transformation` function, since the RACE does no image processing. The master coordinates one or more `Slave` nodes. Slaves are used to buffer large numbers of tiles, and have connections upstream to the pipeline and downstream to visualisation clients.

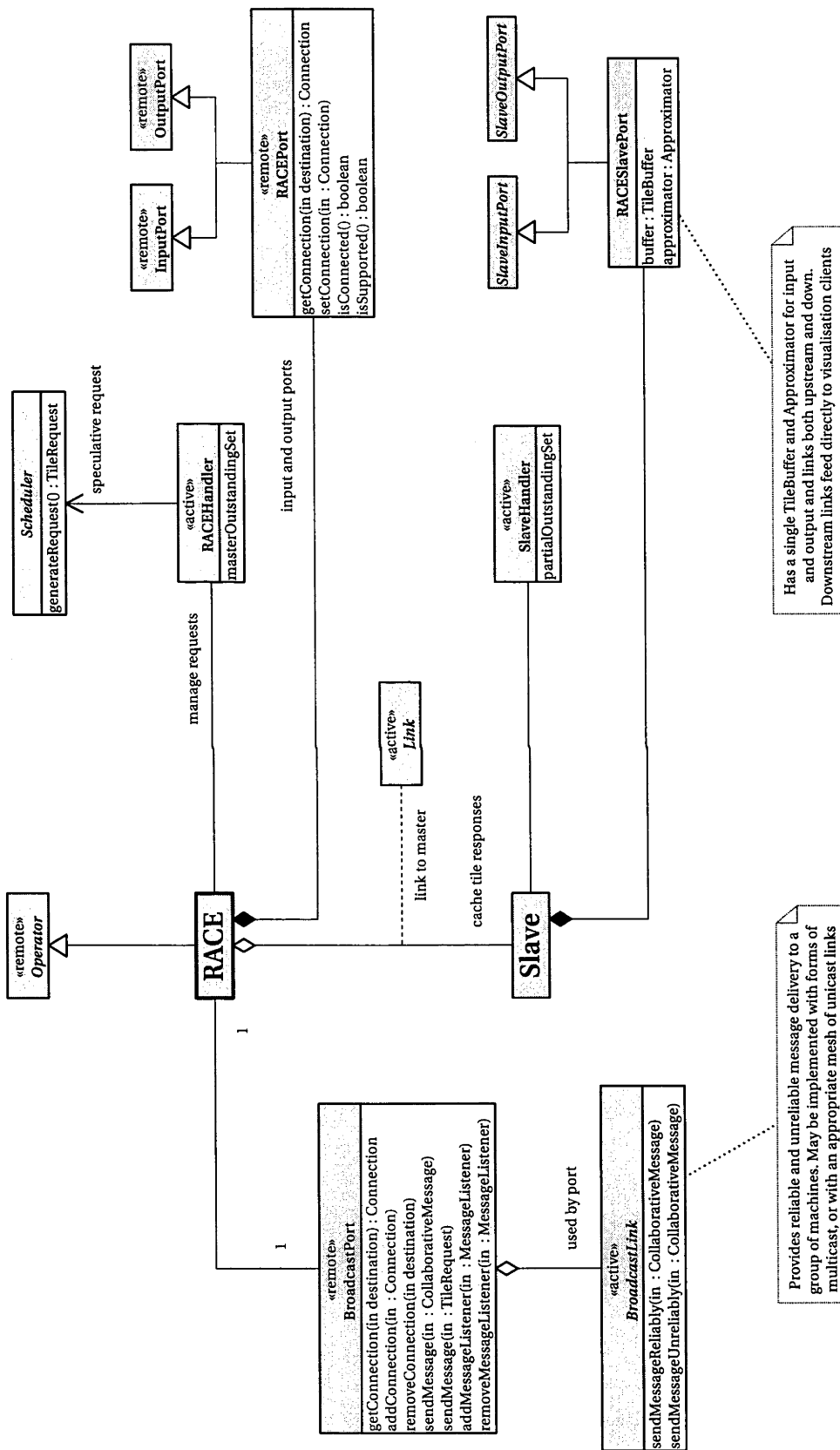


Figure 4.11: Architecture of the RACE as an extension of the standard operator model

#### 4.5.2 Nodes and Ports in a RACE

RACEPort	An operator port on the RACE master node	p. 185
RACESlavePort	A port on a slave node	p. 185
RACEHandler	The request handler on the master node	p. 184
SlaveHandler	The handler on slave nodes	p. 186

Each RACE has a single master node and multiple slave nodes. The design of the RACE master node is a variation of the general `Operator` which uses a highly specialised form of operator port and a greatly simplified request handler. The master also has a `BroadcastPort` used for collaborative data sharing, which is considered separately in section 4.5.5. The master does not actually cache any data. Clients interact with the master node only when first connecting to the operator, and to share collaborative data. All tile requests are directed to `Slave` nodes, who do cache data.

The master node has a `RACEPort` for each of the type of data produced by the pipeline. A `RACEPort` is both an `InputPort` and an `OutputPort`, but it does not buffer tiles. The purpose of the ports on the master is to allow the cache to join the pipeline during the service initialisation phase. When a RACE is connected to the pipeline the `Connection` objects sent to the master's input ports are replicated to the ports of all slave nodes. This allows every slave to build a complete set of links upstream and so exploit upstream scattering. The master's ports are also used to provide a `Connection` object to clients as they join the pipeline. The `Connection` object a client retrieves from the master's `RACEPort` describes how to build links to all `Slave` nodes. In addition to the links upstream, the master node has a `Link` to each `Slave` node in the cache, used for internal communications. Communication on the master's ports is coordinated by a specialised form of request handler, known as a `RACEHandler`. This handler is discussed in detail in the next section.

The design of the `Slave` nodes also a simplified form of `Operator`, lacking a `Transformation` object and with a greatly simplified handling mechanism. Communication to slave nodes is through one or more `RACESlavePort` objects. These serve as both input and output ports, with a single `TileBuffer` and links both upstream and down. The `TileBuffer` knows about, and can snoop, buffers on other `Slave` nodes and may also have an `Approximator` function. Slaves have a `ProxyHandler` to listen to messages from their ports, and relay them on to the `RACEHandler` running on the master node. Each slave has a single specialist `Link` to the master node, not associated with any port, used simply to relay `RAPPORT` messages.

Before describing the request handling behaviour of the RACE, it is important to understand how and where tiles are cached. Large datasets are decomposed into a series of tiles, and tiles are partitioned across the slave nodes in the RACE. The partitioning policy that maps tiles to slaves can operate many different ways, and should be specified in the implementation profile. For example an HPF-style [59] cyclic distribution policy could be used to statically assign tiles to slave nodes. Alternatively, tiles of different types could be stored on different slaves: all imagery on one slave, all elevation data on another. Replication of tiles adds further possibilities for interesting tile partitions. Whatever partition policy is used, it is encoded in the `RequestMap`

on each visualisation client. This allows clients to send `TileRequest` messages directly to the appropriate `Slave` node, and so minimise response time for a cache hit. If the partitioning is not static, clients will need to be informed of changes to the partition through `MapUpdate` messages.

### 4.5.3 Handling Requests

<code>RACEHandler</code>	Request handler run on the master node	p. 184
<code>SlaveHandler</code>	A proxy handler run on slave nodes	p. 186

The request handling behaviour of RACE is rather different to that of other operators. Request handling is simplified by not having to support a `Transformation` function in the RACE, but complicated by the need to support inter-site cache snooping. It is further complicated by the need to support speculative request scheduling. The logic to handle a request is split between the `RACEHandler` on the master node, and the `ProxyHandler` running on each slave.

**Handling Tile Request Messages** `TileRequest` messages are sent from clients directly to the slave nodes of a RACE. No transformation of data is performed by RACE, it simply buffers responses from upstream operators, so requests do not need the same complex support model described for operators in section 4.4.4. Consequently, the handlers running on slave and master nodes need to maintain only the `OutstandingSet` of requests. This is necessary so that a port can send responses to all requesting clients. However, there is no need for a `DependentSet` since there is a many-to-one mapping from downstream requests to upstream requests<sup>34</sup>. There is also no need to maintain supported and output queues since there is no transformation function for the handler to interact with.

When a slave receives a `TileRequest`, the slave's `RACESlavePort` processes the request much as any other `OutputPort` would. If the tile is in the local `TileBuffer` the slave sends a `TileResponse` immediately and the client request is satisfied. If not, the port may produce an approximation before passing the request on to the local `ProxyHandler`. The `ProxyHandler` maintains a local `OutstandingSet` to record all unanswered requests received by the slave. It does not, however, have the appropriate logic to make requests upstream. Instead it passes the request on to the master `RACEHandler` via the master `Link`<sup>35</sup>.

The master node's `RACEHandler` sends `TileRequest` messages up the pipeline. When the master handler receives a `TileRequest` it is placed in a global `OutstandingSet`. The request is then sent to the appropriate upstream operator through one

<sup>34</sup>Unlike transforming operators, where there may be a many-to-many mapping of downstream requests to upstream requests.

<sup>35</sup>Before passing requests to the master, the slave updates the `TileRequest.returnLink` attribute to reflect the name of the link from the upstream operator to the slave. This is required to support upstream scattering, and allows the upstream operator to send the `TileResponse` message directly to the slave.

of the master's input ports. The master will not receive a response to this request directly, since the return link specified in the request is to a slave node. In parallel with the request to the pipeline, the master broadcasts the same request to the RACE running at all other sites via the inter-site (described below). The message is modified in two significant ways before being sent to other caches. First, no approximations are requested, whatever the original request may have allowed. Second, the `doNotPropagate` flag in the request message is set to true to stop other caches sending the same request to the pipeline.

**Response Handling** The semantics of response handling also differ from those of a normal operator, since they depend on where the response came from. If another RACE responds before the pipeline, the response will be sent to the master node. The master `RACEHandler` sends a `CancelRequest` message to the pipeline, removes the request from the global `OutstandingSet` and sends it to the appropriate slave through its master link. If, however, the pipeline responds before another RACE then the response will be sent to the slave node. The slave sends the response on to all clients that requested the tile, removes the request from its local `OutstandingSet` and then sends the response header – *but not the full tile* – to the master. This allows the master to remove the request from the global `OutstandingSet` without the cost of sending the full response payload.

**Cache Snooping** One final subtlety to request handling is required to support inter-site cache snooping. When a master `RequestHandler` receives a tile request from the inter-site bus it sends the request on to the appropriate slave via the master link. If the slave has the tile in its `TileBuffer` it sends a response back to the master, which in turn sends it to the requesting site. If the slave does not have the tile, the request is simply ignored. Inter-site requests are not put in either `OutstandingSet` since no response is guaranteed; a site only responds if it can.

#### 4.5.4 Scheduling Speculative Requests

---

Scheduler	Runs on the master node and makes speculative requests to improve responsiveness and throughput	p. 186
-----------	---	--------

---

The master node uses a request `Scheduler` to make speculative requests for tiles and keep the pipeline running at peak throughput. The role of the RACE `Scheduler` differs somewhat from that of schedulers in other distributed systems. The RACE scheduler is so called because it schedules the flow of tiles through the pipeline *in the absence of user requests*. User demand-driven requests have a higher priority than speculative requests, and are satisfied before requests made by the scheduler.

The `RACEHandler` on the master node determines a rate at which requests should be made. This rate evolves over time, in response to `FlowControl` messages received from the pipeline. The handler uses the request `Scheduler` to produce speculative requests to reach the required rate after all client-initiated requests have been sent.

Speculative requests can be based on any or all of the scheduling policies described in section 3.1.2. Implementing these policies is a complex problem in its own right and a rich source of future research ideas, but is not specified here. It should, however, be a fundamental element of any implementation profile.

#### 4.5.5 Inter-site Bus and Collaborative Data Sharing

BroadcastPort	Inter-site sharing port on the master node	p. 183
BroadcastLink	Link used to form the inter-site sharing bus	p. 184
CollaborativeMessage	Collaborative data sent over the bus	p. 179

When users from more than one site access the same dissemination pipeline it is useful for all the site caches to communicate, to perform cache snooping and to share collaborative data. An inter-site sharing bus is established by providing each RACE with a link to the BroadcastPort of every other RACE. The master node of a RACE has a BroadcastLink to every other master node, which it uses to send CollaborativeMessage and cache snooping TileRequest messages. It also has a link to every client running at the local site, which is used to send and receive collaborative messages for clients. The links and ports used to form the inter-site bus are subtly different from those used in the dissemination pipeline and deserve some consideration.

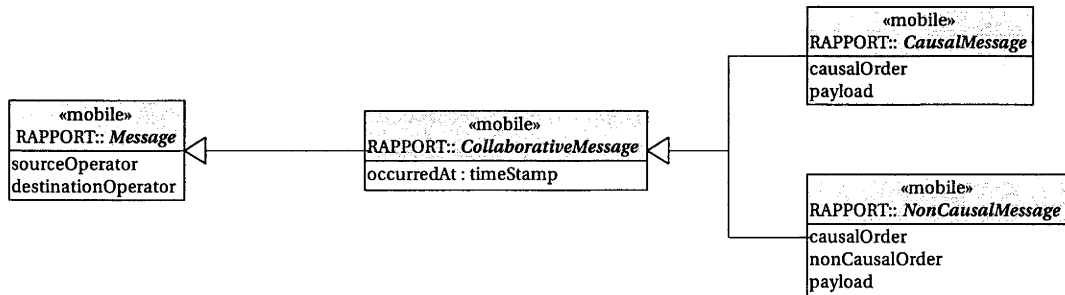
The set of connections used by the inter-site bus is a variation on the *Distributed Peer with Client-Server sub-groupings* [85] topology<sup>36</sup>. Naive implementations of this topology, based on point-to-point network connections, scale poorly with respect to the number of sites and the number of users per site. The problem is made worse by the need to support two different classes of traffic over the bus, for reliable and unreliable delivery, imply that two links are required between each node. Scalability is not a high priority for many implementations; unreliable [23] and reliable [31, 89] forms of multicast may offer a solution when scalability is important<sup>37</sup>. However it is implemented, the essential requirement of the inter-site bus is that it support one-to-many communications, with messages being sent from a single source to multiple destinations.

Several new classes are used in the RACE and in visualisation clients to provide the inter-site bus. Some of these are presented in Figure 4.11. The cache master node and clients all have a BroadcastPort used to send and receive collaborative messages and tile requests. These ports are connected through BroadcastLink objects which provide one-to-many message delivery with reliable and unreliable services. The RAPPORT message hierarchy is extended to include new classes of CollaborativeMessage, as shown in Figure 4.12. Two concrete classes of message are supported: CausalMessage and NonCausalMessage. A CausalMessage has a causal ordering and requires reliable delivery. A NonCausalMessage has causal and non-causal

<sup>36</sup>Please refer to sections 3.1.4 and 3.1.5 and Figure 3.2 for more details of this topology

<sup>37</sup>Multicast is no panacea. Reliable multicast protocols are still largely unproven, and remain an open research problem. Unreliable multicast is a commodity technology, but though very effective it is poorly supported over wide area networks.





**Figure 4.12:** Additional RAPPORT messages to support collaborative data sharing in the RACE

ordering, and can be delivered over an unreliable connection. This provides the minimum support for collaborative data sharing with sufficient causal ordering, as described in section 2.2.3.

## 4.6 Minor Details and Points of Clarification

Having considered the RAPID architecture in detail from service negotiation through to the provision of service, only a few minor details remain.

### 4.6.1 How Clients Connect to a Pipeline

Visualisation clients connect to the pipeline during the service provision phase. They start by looking for the pipeline advertisement in the grid directory service. This returns a global reference to the `PipelineManager` in charge of the pipeline. The client uses the manager's `getClientConnection()` method to join the pipeline. This method returns a reference to the RACE the client should use to interact with the pipeline, make requests and share collaborative data. The client then connects to the RACE just as any other operator connection is formed. Security is not considered in this thesis, but in a secure environment the interaction between client and manager could be used to perform some authentication. This is not a complete solution but it provides a useful starting point.

### 4.6.2 Termination of Service

The `PipelineManager` is responsible for terminating a pipeline after the agreed period of service has elapsed. Termination automatically takes place when the manager notes that the `endServiceTime` has passed, and can also be triggered by external actors by calling the manager's `abortPipeline()` method. The manager is responsible for graceful termination of a pipeline, but resources can still be reclaimed even if a manager does not terminate the pipeline. Operator factories are only bound to provide an operator for the period agreed during service negotiation. Once that

---

period has elapsed, the factory is perfectly entitled to terminate the operator and reclaim the resources associated with it. The operator failure semantics of the pipeline are described in section 3.2 on page 52.

### 4.6.3 Tiling and Level of Detail Mechanisms

No particular Tiling or Level-of-Detail (LoD) mechanism is defined as part of the RAPID pipeline. Neither is a co-ordinate system for registering data sets. These are application specific details and so should be specified in an implementation profile. All the RAPID pipeline offers is a common approach to tiling and LoD mechanisms. The `getTileHierarchy()` method of `OutputPort` returns a description of the tiling and LoD mechanisms used in the pipeline. This description provides enough information to infer the range of acceptable values for the `LoD`, `tileCoordinates` and `tileSize` attributes of the `TileRequest` message. The format of this description should be specified as a key detail of the implementation profile.

### 4.6.4 Observability

Observability is an important property of any large distributed system. The manager supports service observation through a listening interface. This allows an interested third party to observe pipeline events such as clients joining or services starting, and also exceptions such as network or node failure.

## 4.7 Summary

This chapter has described the Responsive Architecture for Pipelined Imagery Dissemination (RAPID) – the major result of this thesis. To satisfy the requirement for throughput, RAPID uses a sophisticated approach to parallel streaming in a dissemination pipeline. To satisfy the requirement for responsiveness, RAPID develops a distributed data cache known as the RACE. The RACE also meets the requirement for collaborative data sharing by providing the backbone of an inter-site sharing bus and is the centrepiece of the architecture. Finally, RAPID satisfies the requirement for application management by defining actors, a schema and standard life-cycle for dissemination pipelines formed within a computational grid.

RAPID is a design-time solution, and as yet there is no single, complete implementation of it. The chapters that follow are case studies into partial implementations. Each study focuses on one of the fundamental requirements outlined in Chapter 1. The next chapter considers the responsiveness requirement. Chapter 6 considers ways to achieve high throughput in a dissemination pipeline and Chapter 7 considers application management. Taken together these studies provide a substantial proof of concept for RAPID.



---

# Comet: a Study in Client Responsiveness

---

In this chapter we move from design to implementation and consider the *responsiveness* requirements of a Virtual Environment as it accesses and uses geospatial imagery. The chapter is a case study into a digital terrain visualisation system, known as Comet [137]. Comet was developed for the tenth plenary meeting of the Committee of Earth Observation Scientists, and subsequently presented to a range of other users of geospatial imagery. As with the other case studies, Comet pre-dates the RAPID architecture, and is not an exact subset. Consequently it serves to both validate and motivate elements of RAPID. In this study it serves as a vehicle for considering client responsiveness and the role of a site cache. In particular it addresses:

- the interaction between visualisation clients and a site cache;
- the use of downstream gathering to optimise site cache request processing;
- the patterns in which clients access data and the implications for tiling and level-of-detail mechanisms; and
- the importance of asynchronous communications.

The purpose of this study is to review the design of Comet, compare it to RAPID and evaluate its performance. Section 5.1 provides an overview of Comet, and section 5.2 discusses some of the unusual requirements that affected its design. Section 5.3 presents the architecture of the application in detail, with particular attention to the threading and caching techniques used on the client, and the approach to managing data on a server cluster. Section 5.4 provides experimental and analytical evaluation of the performance and responsiveness of Comet. A key goal of this evaluation is to determine how clients interact with a dissemination pipeline and a site cache. Finally, section 5.5 presents a summary of the results and experience gained from the Comet project. This organisation is repeated in the other case studies in subsequent chapters.

## 5.1 Overview of Comet

Comet is an interactive visualisation tool, which allows users to explore very large Digital Terrain Models (DTMs) in real time using commodity hardware. It was designed specifically to showcase geospatial imagery from the IRS-1 family of satellites, operated by the Indian Space Research Organisation.

In November 1996 Comet was demonstrated at the tenth plenary meeting of the Committee of Earth Observation Scientists (CEOS) held in Canberra, Australia. The demonstration involved a real-time visualisation of a 446-Megabyte digital terrain model of Canberra and the surrounding Namadgi National Park. This DTM was a fusion of two different sets of IRS-1 imagery with an elevation model. It was produced using mass-storage and high performance computing resources at the Australian National University in Canberra and the University of Adelaide [57, 58]. These resources, which are physically separated by 1100km, were connected via the Telstra Experimental Broadband Network<sup>1</sup> (EBN) [162]. Figure 5.1 shows the raw datasets that were fused to produce the final DTM. For visualisation purposes the DTM was maintained on a cluster of Digital Alpha workstations at the Cooperative Research Centre for Advanced Computational Systems (ACSys), and distributed across the city to the visualisation client by a metropolitan area ATM network. Figure 5.2 presents images of various interesting landmarks in the Canberra region, taken during the demonstration.

The CEOS demonstration was a complete working example of how to disseminate and visualise large geospatial datasets in real time. It involved large imagery archives, high performance computations and broadband networks. However, much of the processing and dissemination of the source datasets, and all application management, was performed by hand. These issues were the motivation for the vGrid and CROP systems presented in later case studies. Comet is not a complete implementation of RAPID, but is a useful vehicle for considering responsiveness.

## 5.2 Requirements of Comet

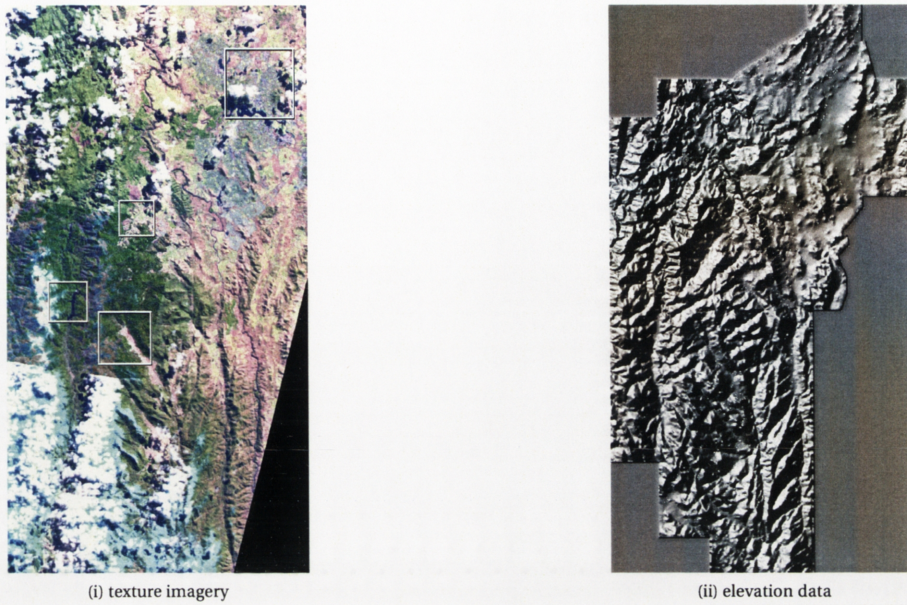
In general the requirements for Comet are representative of most visualisation systems. However, there are three notable aspects to Comet's design: it is intended specifically for a single type of geospatial imagery; it compromises some responsiveness in the interests of higher fidelity rendering; and it is required to inter-operate with legacy software. These requirements do not undermine the value of Comet as an experimental platform, but add interest to this case study.

Comet is intended solely to render imagery from the IRS-1 family of satellites. These are equipped with a low-resolution (24 metre) multispectral sensor that produces false colour images, and a higher resolution (6 metre) panchromatic<sup>2</sup> sensor.

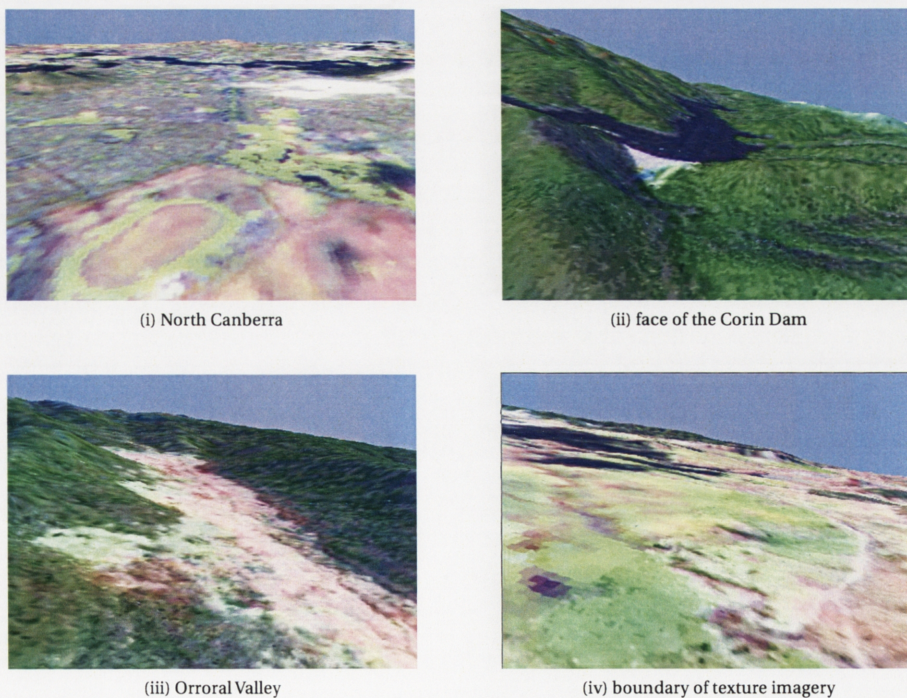
---

<sup>1</sup>Australia's first broadband network which provided 155mbps links between the two sites with average round trip times on the order of 50 milliseconds

<sup>2</sup>Which produces grey scale/monochrome imagery.



**Figure 5.1:** Datasets used to produce the Digital Terrain Model for the CEOS demonstration: texture imagery (i) was a fusion of IRS-1C LISS (colour) and PAN (greyscale) images; elevation data (ii) was provided by the ACT government. The squares in (i) indicate the locations of pictures in Figure 5.2



**Figure 5.2:** Images produced by Comet during the CEOS demonstration: (i) looking south toward the city of Canberra with the race course in the foreground; (ii) the face of the Corin Dam; (iii) the Orroral Valley; and (iv) the boundary between the two sets of IRS-1 texture imagery

An unusual feature of the satellites is the ability to produce stereoscopic pairs of images, which can be used to produce high-resolution elevation models at the same resolution as the monocular texture images. To properly demonstrate this feature the Comet renderer assumes that the elevation data in a terrain model is available at the same spatial resolution as the texture imagery. This is a relatively unusual situation and means that in some respects Comet is more like a voxel-based renderer than the polygon-surface renderers described in section 2.2.4. It also means that texture and elevation data are distributed together in Comet, where other applications might choose to distribute them independently.

Another notable aspect of Comet is the way it compromises some user responsiveness to produce a high quality visual output. As described Chapter 2, most Virtual Environments attempt to redraw the display at a rate of 60 frames per second. Comet renders at a lower rate (5-15 frames per second depending on the visualisation hardware) in return for higher quality visual results. For this reason frame-rate is not a good measure of performance for Comet. Many of the experimental results, presented later, are measured in terms of total render times rather than average frame rates.

A final notable aspect of Comet is that it must interact with a set of legacy software, which requires communication through the Sun RPC [103] mechanism. Network links used in Comet are implemented over RPC and so are inherently synchronous in nature. This is an important difference to the links used in RAPID. Threads are used extensively to overcome this limitation and make communications more asynchronous.

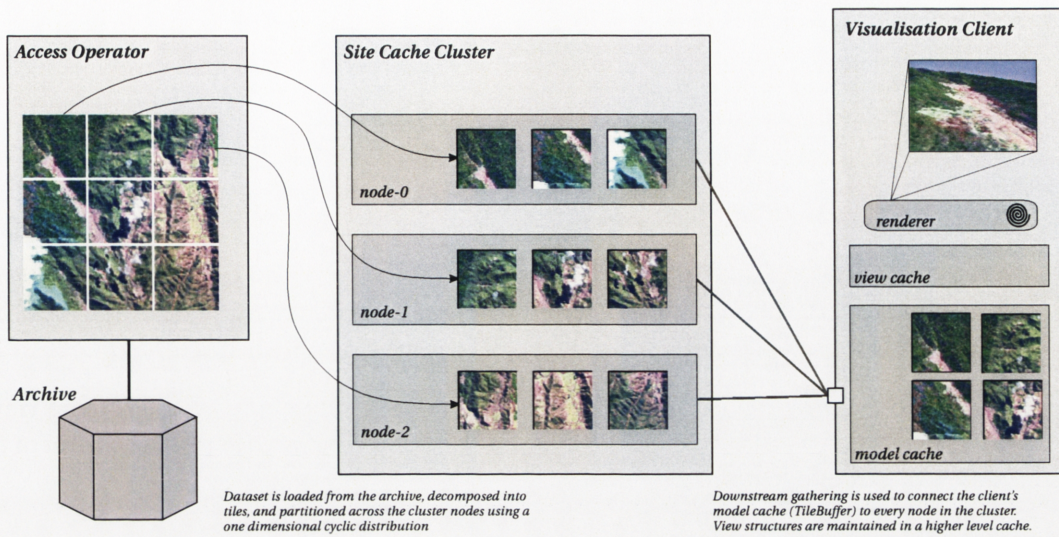
## 5.3 Architecture of Comet

Although Comet pre-dates RAPID it contains many of the important ideas and validates much of the architecture. Figure 5.3 depicts the major components of Comet. This includes a visualisation client and a RACE-style parallel site cache. The site cache runs on a cluster of server machines, and is used as an in-memory store for large datasets. Tiles are streamed from the cache to clients using downstream gathering. Clients also use a form of speculative fetching by Request Proximity, described in section 3.1.2, to mask latency.

Despite these similarities there are important differences between the two designs, and limitations of Comet serve to motivate parts of RAPID. We will now briefly review four important aspects of Comet: the use of a parallel site cache; the operation of the client renderer; client-side caching; and asynchronous communications techniques.

### 5.3.1 Data Distribution and the Server Cluster

Since it is not practical for clients to hold a complete copy of a large dataset, Comet stores datasets in a large parallel site cache, on a server cluster. This cluster is equiv-



**Figure 5.3:** The architecture of Comet. The site cache is built on a server cluster and is accessed by one or more visualisation clients. The cluster maintains an in-memory cache of tiles of geospatial data. Clients request only those tiles of data visible from the user controlled camera.

alent to the RAPID Cache (RACE)<sup>3</sup>. Communication between clients and the cluster do not exactly match those described for RAPID, because the underlying network links (equivalent to the RAPID `Link` object) are based on RPC. This forces interactions between clients and the cluster to be completely synchronous. In other respects the interactions between client and cluster are very similar to those of RAPID.

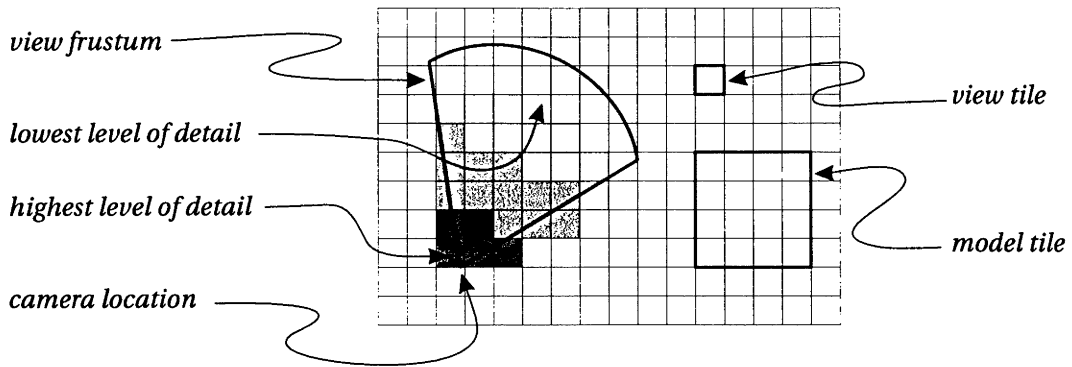
The data cached in the cluster is a fusion of texture and elevation data. Consequently only a single dataset is exported from the cluster, shared through a single port. The dataset is tiled, but only one tile size is supported and no level of detail mechanism is used. The tiles of data stored in the cluster are known as *model tiles*, since they correspond to a model-level structure rather than a directly renderable view structure<sup>4</sup>. The implications of using a single fixed model tile are considered later in this chapter.

Since the cluster is a parallel cache, downstream gathering is used to stream tiles directly from each node in the cluster to clients. Downstream gathering requires that clients have a request map function (a `RequestMap` object) to send requests to the correct node in the cluster. This mapping is easy to perform because datasets are partitioned across the cluster using a static, one dimensional cyclic distribution. Consequently, clients are able to directly translate the coordinates of a model tile to

<sup>3</sup>In this chapter the term “cluster” is used to refer to the site cache or RACE component. Comet uses two caches on the client side in addition to the cluster cache. Rather than overload the term cache, the term cluster is used to refer to the site-wide cache.

<sup>4</sup>See sections 2.2.1.1 and 2.2.4 for more details.





**Figure 5.4:** Determining visible tiles and level of detail. The renderer uses the camera view frustum to determine what view tiles need to be drawn and the distance from the camera to determine level of detail.

a link to the appropriate cluster node.

### 5.3.2 The Rendering Process

Terrain rendering drives the execution of the Comet client. The actual renderer is relatively simple, and highly specialised to the imagery from the IRS-1 family of satellites. For example, the renderer does not use a continuous level of detail algorithm, such as those described in section 2.2.4, but a discrete LoD mechanism. However, its interaction with other elements of Comet is representative of more sophisticated terrain renderers and relevant to the general problem.

Like the cluster, the renderer treats the terrain surface as a uniform grid of tiles, known as *view tiles*. For each frame it determines the set of view tiles that are visible given the location and orientation of the user controlled camera. This process is shown graphically in Figure 5.4. It is important to note that the size of the tiles used by the renderer need not be the same size as the model tiles used for distribution. Indeed to avoid unnecessary overdraw, and visual artefacts such as “pop-up”, the renderer typically uses much smaller tiles than those held in the cluster. So there is an important distinction between the model tiles described in 5.3.1 and the view tiles used by the renderer. No view tile ever overlaps the boundary of two model tiles, so there is always a direct mapping from a view tile to a single underlying model tile.

Once the visible set of view tiles is known, each tile is rendered at an appropriate level of detail. The renderer uses six discrete levels of detail to represent view tiles at different distances from the camera. The rendering of each individual tile is performed using OpenGL display lists, where each list contains the geometric and texture description of a single view tile at a single level of detail. The preparation and management of these lists is a non-trivial task and is performed by the view cache, as described below.

---

### 5.3.3 Client Caching Strategies

The tiles required to render two consecutive frames are overwhelmingly the same. Client-side caching is therefore a very useful way to improve rendering performance. Comet uses two levels of client caching: a low-level cache that contains raw terrain data (model tiles) from the cluster; and a higher-level cache that contains directly renderable data structures (view tiles).

Each client maintains a low-level cache of model tiles in a tile buffer. This is exactly like the `TileBuffer` on any input port of a RAPID operator. The buffer, or *model cache*, is arranged as a direct-mapped two-dimensional array of model tiles retrieved from the cluster. Approximations are not supported by the buffer (i.e. it does not include an `Approximator`). Tile requests that cannot be satisfied from the buffer automatically trigger a synchronous network request (RPC) to the appropriate node in the cluster.

A second level of caching is used on clients to manage renderable tiles. This is known as the *view cache*, and it manages an associated set of OpenGL display lists, accessed through a direct map. Each list represents a single renderable view tile at a single level of detail. Because of the possible discrepancy in size between model and view tiles, one model tile may map to several tiles in the view cache. The renderer requests an appropriate display list from the view cache, for each view tile required for a frame. When a renderer request causes a view cache miss, the view cache automatically requests appropriate data from the underlying model cache. This may in turn lead to a model cache miss with an associated request sent to the server cluster. When both caches miss the rendering thread blocks while an RPC is made to the cluster. Once the raw model data is available, it is sub-sampled, tessellated to form polygons, textured and turned into an OpenGL display list ready for immediate rendering. The cost of these operations is not insignificant: one reason for having a separate view cache.

Additionally, the view cache holds elevation information for the corners of all tiles in the digital terrain model. This allows the renderer to perform tile visibility tests to determine what view tiles should be rendered without requiring all candidate tiles to be in either cache.

### 5.3.4 Multithreading and Quasi-asynchronous Data Delivery

In the worst case a simultaneous view and model cache miss can cause the rendering thread to block while waiting for a tile from the cluster. To try to mask the cost of model cache misses the Comet client uses an aggressively multithreaded approach to fetching data from the cluster, independently of the rendering thread. This approach attempts to ensure that when requests from the render thread cause a view cache miss the request can be satisfied by the model cache, and so avoid exposing the render thread to the network latency.

At the start of each frame the render thread determines the set of tiles within the view frustum of the camera. This set is ordered so that tiles present in either

client cache are rendered before those that must be retrieved from the cluster. While the rendering thread is occupied with local tiles, a pool of fetch threads is used to request the missing model tiles from the cluster. The aim is to overlap requests for remote tiles with the rendering of tiles already in the local cache. This decouples the rendering thread from data fetch duties and has the effect of making the otherwise synchronous requests quasi-asynchronous. The more fetch threads used the greater the number of cache misses that can be handled asynchronously.

The limitations of this technique are twofold. First, aggressive multithreading inevitably introduces synchronisation overheads. However, since model tile requests are independent the fetch threads require minimal synchronisation. Second, this technique can only isolate the render thread from network latency when the client-cluster request-response time is less than the time to render the cached tiles. This is true for a useful range of client-cluster latencies: local and metropolitan area networks (LANs and MANs) are typically quite fast enough.

## **5.4 Analysis of Comet**

Comet provides an opportunity to explore the interactions between visualisation clients and a RACE-style site cache. In particular it allows us to consider the effects of cache size and latency (from either the network or a dissemination pipeline) on client responsiveness. It highlights the importance of asynchronous communications, tiling and level of detail mechanisms and provides a way of quantifying bandwidth usage. Finally, Comet uses a basic implementation of the Request Proximity prefetch heuristic, described in section 3.1.2, which is also evaluated.

### **5.4.1 Test Environment and Metrics**

The hardware infrastructure used to test Comet was the same as that used for the CEOS demonstrations<sup>5</sup>. Several different networks were used in testing. The primary network was a 155mbps ATM LAN that coupled the Alphastations. Latency within the LAN was negligible – a fraction of a millisecond – and applications were able to sustain a bandwidth of over 16 Megabytes per second. The ATM LAN was augmented with a basic 10mbps switched Ethernet network. Although numerous national and international broadband connections were also available they were not used in the testing process. Higher network latencies were simulated and the wide area networks used to validate the results of simulation.

The simulation of network latency affects the performance results that are reported in this section. Frame-rate is perhaps the best measure of performance for

---

<sup>5</sup>It consisted of a cluster of eight Digital Alphastation 600 machines with 266MHz EV5 CPUs, 128MB RAM running Digital Unix 4.0. Since Comet is portable across a range of architectures and operating systems, alternative PC-based clients were also used in some tests. These machines included a dual processor Celeron-400 based system with a Matrox G400 graphics board and a single processor Celeron-300 with an nVidia TNT graphics card, both running Windows NT 4.0 SP5.

---

visualisation systems, but it is not reported here for two reasons. First, as already described in section 5.2, the average frame-rate of Comet was deliberately compromised in the interests of visual fidelity. Second, frame-rate is highly sensitive to jitter in the network, and since network behaviour was simulated variations in rate would essentially be artefacts of the network simulation.

To avoid these problems, application performance is reported in terms of total render time for a well known exploration of a dataset. Several different datasets were used during testing, with a range of different exploration paths through each set. Each exploration requires the client to render an exact sequence of frames along a predefined path. Application performance was measured as the total amount of time required to render all frames of a given exploration.

### 5.4.2 Caching

Client-side caching is used to isolate the renderer from network latency and improve responsiveness. Independent tests were performed to consider the effectiveness of both the view cache and the model cache used by the client. These results demonstrate how client caches partially decouple the rendering processes on the client from the dissemination pipeline. The model cache results have additional significance since they also apply to the design of a site cache (RACE) and its interactions with a dissemination pipeline.

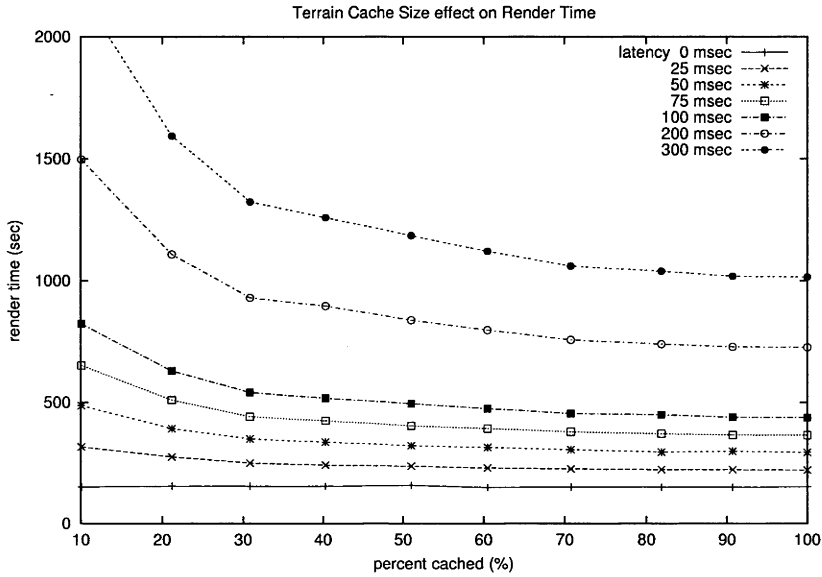
The importance of client-side caching is evident from the hit rates of the two caches. On average the view cache is able to satisfy 99.06% of tile requests from the renderer<sup>6</sup>. Performance of the model cache is less remarkable: on average 58.5% of requests from the view cache are satisfied by the model cache. In large part the relatively poor performance of the model cache is a result of the overwhelming success of the view cache. But the view cache has limits. Other results, not presented here, indicate that increasing view cache size is only valuable to a point. Having very large view caches does not help performance. The optimal size of the cache should be derived empirically since it is a complex function of tile size, dataset size and client display resolution. What is important is that the client needs to cache only a small fraction of a dataset in renderable data structures.

The effect of model cache size on performance was measured experimentally. These results are relevant to both the client's interactions with a site cache, and also to the cache and its interactions with a dissemination pipeline. The critical issue is how latency affects client performance. To measure this, the model cache size was varied as a percentage of the total dataset and total render time was recorded for a range of latencies. Separate measurements were made for both hot and cold caches. A hot cache is one which is pre-loaded with model tiles, while a cold cache is initially empty. The results of this test are presented in Figure 5.5.

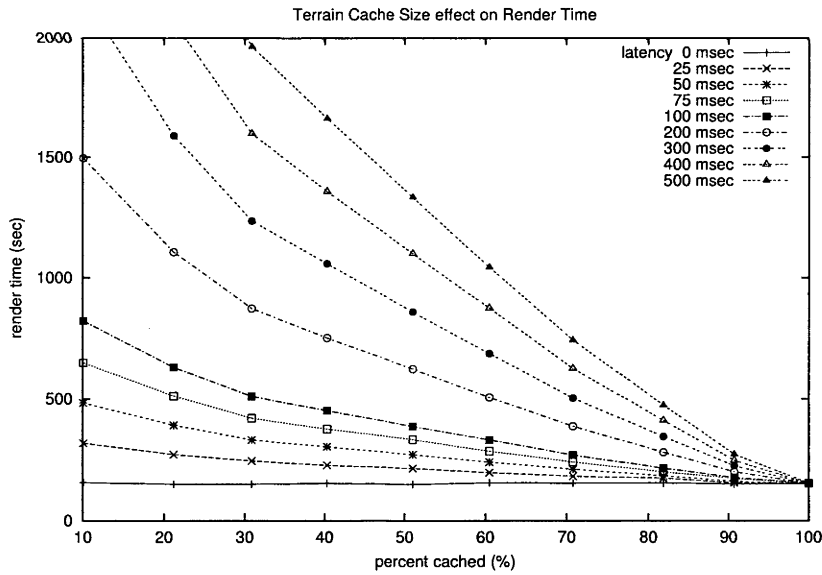
The cold cache results in Figure 5.5(i) indicate that very large client model caches

---

<sup>6</sup>The standard deviation from this average is 2.6%, and this results almost entirely from the first few frames where the view cache is empty.



(i) cold cache performance



(ii) hot cache performance

Figure 5.5: Effectiveness of (i) cold and (ii) hot model caches. Performance as a function of model cache size for a range of network latencies. Performance is measured in terms of total render time, cache size as a percentage of the total DTM.

---

do not solve all client performance problems. In all cases there is a marked performance improvement up to a 30% cache size, but beyond this point the return on larger cache size becomes marginal. This result supports the conclusion that a minimum client cache is required for high rendering performance but larger caches are not beneficial. The reason large caches add so little is due to the cost of cache misses. With a cold (empty) cache there will always be a miss the first time a tile is requested. Even a 100% cache must be loaded with data, and that defines a lower bound on performance for a purely request-driven system. As latency increases the first-miss cost kills performance even with very large caches. This motivates the use of the speculative fetching and pipeline scheduling techniques described in section 3.1.2.

The hot cache results demonstrate the scope for improvement when speculative fetching is used. A hot cache does not incur the initial miss cost of a cold cache, though it can still thrash. The results in Figure 5.5(ii) show again that there is a discernible performance improvement up to a 30% cache. However, unlike the cold cache there continues to be a near-linear improvement up to a 100% cache, where all curves obviously converge. This result is highly relevant to the design of the site caches, such as the RACE. Clearly site caches should be as large as possible and should be pre-loaded or should speculatively fetch data. When this is done, a large site cache can effectively isolate clients even with very high pipeline latencies.

Finally what is obvious from both the cold and hot cache results is the direct relationship between performance and network latency. Clearly network latency is the major limit to application performance. This motivates the use of techniques for request aggregation and asynchronous data fetch.

### 5.4.3 Tile Sizes for Rendering and Distribution

Request aggregation is a common technique for improving performance of distributed systems. The number of requests made by the Comet client is dependent on the size of the view and model tiles it uses. Using larger model tiles than view tiles is a form of request aggregation, since multiple renderer requests are satisfied in a single network request. However, it is also speculative in the sense that a model cache miss may only satisfy a single view cache miss. So this is one simple way to implement the Schedule by Request Proximity prefetch policy.

Experiments were run to determine how effectively model tile size can be used for request aggregation. View tile size must be chosen to optimise rendering performance and minimise visual artefacts. For Comet the optimal view tile corresponds to 16k of model data. Model tiles can be significantly larger than view tiles. To determine optimal model tile size, client performance was measured for five different sizes over a range of client-cluster latencies. Since network latencies had to be simulated the application performance was measured in terms of total render time, to avoid artefacts of simulated network jitter. Model tiles must be at least as large as view tiles, so model tile size was varied from 16K up to 4 Megabytes<sup>7</sup>. With large

---

<sup>7</sup>Since the tiles are two dimensional and the size is doubled in each dimension, this corresponds to

---

Tile size (kilobytes)	Transfer time (milliseconds)
16	0.97
64	3.88
256	15.53
1024	62.14
4096	248.55

---

**Table 5.1:** Approximate transfer times for different sizes of model tile on a 155mbps ATM LAN

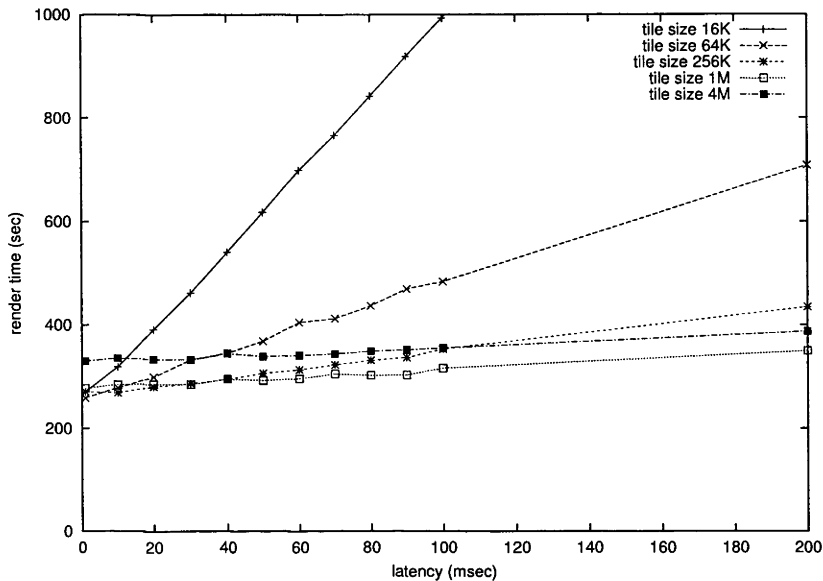
tiles the bandwidth of the network affects the time required to transfer from cluster to client. Approximate transfer times over a 155 mbps ATM network for each tile size are summarised in Table 5.1. The results of the experiment are presented in Figure 5.6.

These results reveal some interesting trade-offs. With negligible latency the small tiles perform best, but as latency increases the larger tiles become more efficient. At low latencies the round trip time for a pair of request-response messages is minimal, and so the tile transfer time becomes the limit to performance. Hence small tiles with an insignificant transfer time out-perform large tiles with significant transfer times. However, at high latencies the request-response overhead equals or exceeds the tile transfer time even for large tiles. In this situation the request aggregation achieved with large tiles comes into play. The reduced number of requests required for a given spatial extent offsets the greater cost of transferring a large tile. This result can be restated more simply. Small tiles work best at low latencies because they minimise the cost of a model cache miss. Large tiles work best at high latencies (where a cache miss is expensive) because they reduce the number of model cache misses.

There is a limit to the use of larger tiles. The results in Figure 5.6 show that for latencies greater than 40 milliseconds the 1-Megabyte tiles outperform all other tile sizes, including the 4-Megabyte tiles. Even with a 200-millisecond latency the 1-Megabyte tiles continue to perform best. Several factors contrive against the very large tile size: cache utilisation and view extent being the principal culprits. Along the edge of the view frustum only part of the model tiles will be used for view tiles. With 4-Megabyte tiles the entire camera view frustum can fit within a very small number of tiles, and so there is considerable amount of unused data in the edge tiles. This data has a negative impact on model cache utilisation as well as requiring that unnecessary data be fetched. This reveals the limit of using model tile size to aggregate view tile requests. Cache utilisation is considered in more detail in sec-

---

increases in the tile size in powers of four.



**Figure 5.6:** Effect of model tile size on performance. Performance in total render time is reported for five different model tile sizes against a range of client-cache round trip times

tion 5.4.5.

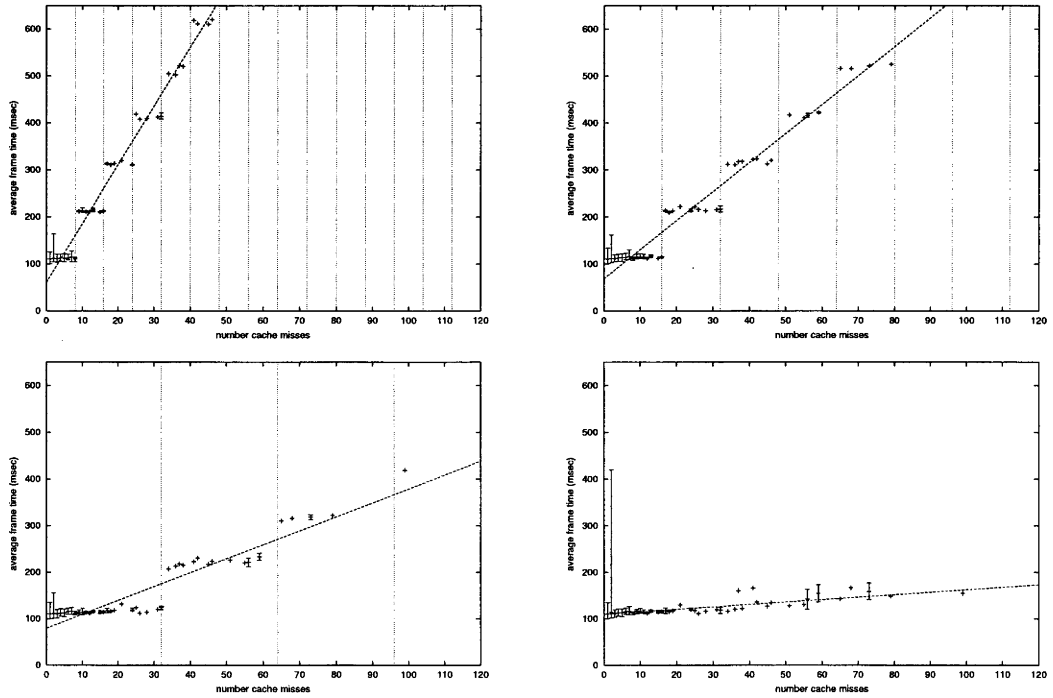
This limit notwithstanding, request proximity is a very useful heuristic for speculative fetching data. It works well when clients access a site cache, and is equally applicable to site cache interactions with a dissemination pipeline.

#### 5.4.4 Multithreading and Asynchronous Communication

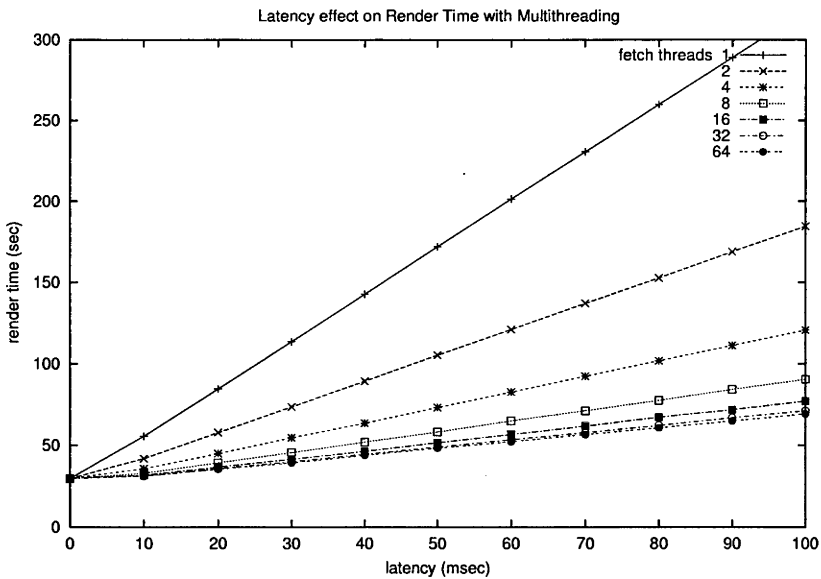
The significance of asynchronous interactions in a dissemination pipeline was also measured experimentally. Comet makes aggressive use of multi-threading to try and decouple the renderer from the cache cluster. Model cache misses are the significant event: tests were run to determine how well the fetch threads are able to isolate the renderer from network and pipeline latency when a cache miss occurs. Figure 5.7 demonstrates the impact of model cache misses on average frame rate. To accentuate the trend these results were produced with a 100-millisecond request-response latency. The results show a clear step function where increasing the number of fetch threads also increases the step size. Linear regressions show the general trend: increasing the number of fetch threads greatly reduces the average frame rate where there are significant numbers of model cache misses. In other words, as latency increases the more asynchronous requests the better. Figure 5.8 shows this trend in terms of total render time, across a range of round trip time latencies. Again, increasing the number of fetch threads can dramatically improve performance in higher latency network environments.

These results highlight the need to decouple execution of visualisation clients





**Figure 5.7:** Effect of multithreaded data fetch. Average frame rate was measured as function of model cache misses with different numbers of fetch threads: eight threads (top left), 16 threads (top right), 32 threads (bottom left) and 128 threads (bottom right)



**Figure 5.8:** Summary of effectiveness of asynchronous data delivery. Total render time as a function of client-cluster round trip latency for various numbers of fetch threads

---

from the rest of the pipeline. The asynchronous behaviour of the RAPID pipeline does this more effectively than aggressive multithreading of remote procedure calls.

#### **5.4.5 Level of Detail and Cache Utilisation**

Although the client view cache and renderer use a level of detail mechanism, the model cache and the cluster do not. Consequently a renderer request for a single view tile at low resolution can cause the delivery of a much larger model tile at full resolution. When this happens unnecessary data is transferred and cached, resulting in poor utilisation of the model cache. This in turn has implications for how much of the dataset can be viewed at once.

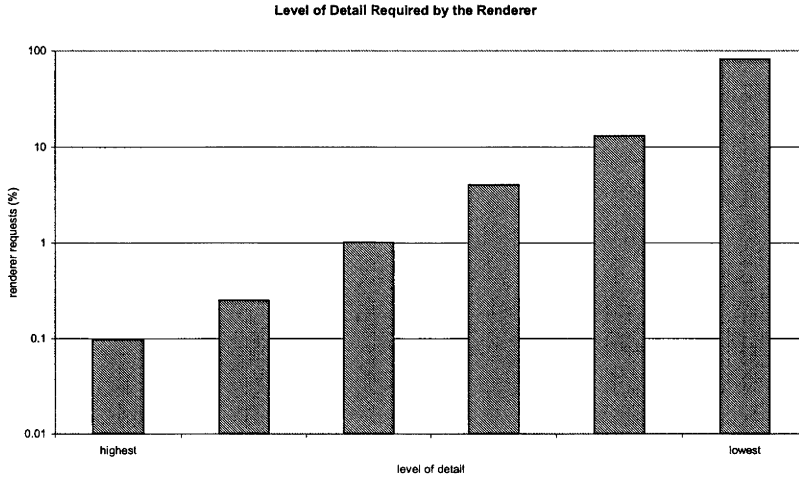
Profiling the levels of detail used by the renderer reveals an important result: the great majority of a dataset is only ever viewed at the lowest levels of detail. Figure 5.9 shows the average level of detail requested by the renderer from the view cache per frame. For an average frame only a tiny fraction of all tiles will be drawn at the highest resolution. The overwhelming majority of requests are at the lowest possible level of detail. Figure 5.10 demonstrates how this impacts on the overall requirements of resolution within the DTM. After an extensive exploration of a DTM, just 2% of the model was viewed at full resolution and 10% was viewed at half resolution, while 71% was viewed at the second lowest resolution and over 99% at the lowest. Clearly, caching all model tiles at full resolution is unnecessary and inefficient.

Model cache utilisation was measured to determine just how inefficient the use of full resolution model tiles is. On average, model cache utilisation was roughly four percent. The navigation path through a dataset obviously has a marked impact on cache utilisation. However, in low-level explorations intended to maximise model cache utilisation, on average only 4.24% of data in the cache was actually required by the renderer. For a 13.2MB cache only 572KB of data was actually required to render an average frame: the rest of the cache was consumed by data at a higher resolution than required.

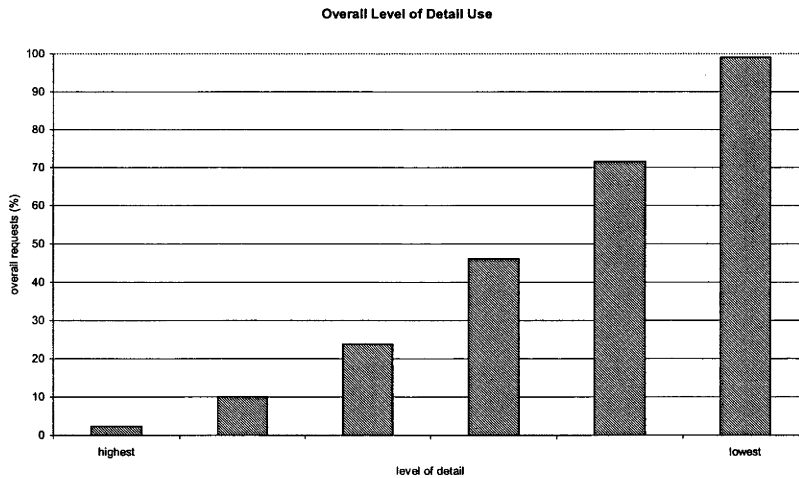
Unfortunately this problem also means that the size of the model cache effectively limits the extent of the dataset that can be viewed at once. When the view frustum of the camera covers an area larger than can be held in the model cache, the cache starts to thrash. At this point performance and responsiveness are unacceptably degraded. This situation occurs when the camera is greatly elevated above the terrain surface, so as a practical matter the size of the model cache limits the maximum height from which the image can be viewed. Since the camera requires only a low level of detail when greatly elevated, model cache thrashing occurs when cache utilisation is at its lowest. The cache thrashes as it attempts to fetch data, the overwhelming majority of which will not be used. This clearly demonstrates the need for a level of detail mechanism to be used between visualisation clients and a site cache.

#### **5.4.6 Network Bandwidth Usage**

It may seem reasonable to imagine that visualising massive images would be a band-



**Figure 5.9:** Average level of detail of view tile requests by the renderer. Note the logarithmic scale of the vertical axis. Only a minute fraction of renderer requests are for the highest resolution. The overwhelming majority of tiles are drawn at the very lowest levels of detail



**Figure 5.10:** Overall access to data at different levels of detail. At the end of an extensive exploration on a very small percent of the data was every required at full resolution

---

width limited problem. In fact this turns out not to be the case, and the bandwidth requirements for Comet clients are easily satisfied by an ordinary local area network (LAN). Analysis of bandwidth use shows that on average Comet requires only 1.187 mbps throughput, with peak use an order of magnitude higher at 11.055 mbps<sup>8</sup>. This leads to the simple conclusion that network latency, and not bandwidth, bound the performance of Comet. It also explains why large client caches aren't especially helpful; there simply isn't that much data to cache. Finally, it highlights the need for scheduling and speculative fetch policies to maintain high throughput in a dissemination pipeline. By itself, demand from a single client is not sufficient to saturate a high-capacity pipeline: even with multiple clients accessing a pipeline the average demand is not great.

## 5.5 Summary

Comet is a system for visualising imagery from the IRS-1 family of satellites. In this chapter it was used as the basis of a case study into client responsiveness. Comet validates many important ideas from the RAPID architecture, and motivates others. The most important results for the case study were:

- Latency, not bandwidth, is the major limit to client performance.
- Use of large, site caches and parallel streaming with downstream gathering was extremely effective.
- Site caches should be as large as possible. Visualisation clients should also cache view and model data structures, but do not need large caches.
- Site caches should use speculative fetch policies to load data and preempt future client requests. Schedule by Request Proximity is a valuable heuristic.
- A level of detail mechanism must be used between clients and a site cache, otherwise client cache utilisation is poor and cache size restricts the visible area of a dataset.
- Asynchronous communications are essential for responsive visualisation. Multithreading offers a partial solution when links are based on a synchronous network protocol, such as RPC.

The experience of the CEOS demonstration also motivated the development of integrated dissemination pipelines and application management frameworks. High throughput dissemination systems are considered in the next chapter. Application management is the subject of the Chapter 7: a case study into the vGrid system.

---

<sup>8</sup>This usage level is partly related to the relatively low frame rates of the renderer, and is also dependent on the speed of exploration of the data. Obviously a high-speed zoom across the surface of the image exposes more data than a slow pan. Balancing this, it is also important to note that the very poor cache utilisation described in section 5.4.5 means much more data is transferred to the client than is strictly required. If even a simple LoD mechanism were used by the model cache and cluster it would dramatically cut the peak bandwidth requirements. So the peak requirement is well within the performance range of a LAN.



---

# Short Studies in Dissemination and Throughput

---

This chapter considers how high rates of *throughput* can be achieved when processing and distributing geospatial imagery. It takes the form of three short case studies into systems developed by the author and other members of the Online Data Archives (OLDA) Program at the CRC for Advanced Computational Systems (ACSys), following the experience of the CEOS demonstration<sup>1</sup>.

A common theme in each of the studies is overcoming the performance limitations of distributed-object middleware. Object Request Broker (ORB) systems, such as CORBA [105], Java RMI [133] or COM+ [62], provide a high level abstraction for network communications by making remote services appear to be part of a local computation. This abstraction allows for simple client development and encourages reuse of services. Many geospatial imagery archives provide search interfaces through an ORB. Implementations of RAPID are likely to use an ORB for the application management components and service negotiation interface. However, ORBs have well-known problems with bulk data transfers [42, 43, 140]. The systems presented here overcome this problem by using specialist bulk data transfer mechanisms connected to form dissemination pipelines.

The three systems are presented in chronological order. Section 6.1 describes the OATS system, which provided a browsing interface for a repository of GMS-5 data. OATS was one of the first-generation of dissemination systems developed by the OLDA team and consequently had some significant limitations. The successor to OATS was a pipeline architecture developed at the Australian Defence Science Technology Organisation (DSTO). It is described in section 6.2 along with experimental results which quantify the performance problems associated with using ORBs. This work had a significant impact on the design of CROP, the third and final system considered in section 6.3. CROP was a comprehensive imagery processing and dissem-

---

<sup>1</sup>The author made a significant contribution to each of these projects. I was responsible for all the design and 75% of the implementation of OATS [58]. I undertook all the performance experiments and analysis presented in section 6.2.2 and 100% of the design of the IMAD-1 architecture [140]. Design of the throughput-related elements of CROP drew extensively on my work with IMAD-1, and I provided input and critical review throughout the project.

ination system, which achieved very high levels of performance.

Collectively these three systems feature many of the major elements of the RAPID pipeline, and demonstrate how to achieve high rates of throughput. They also highlight the need to perform application management for large dissemination pipelines; a problem considered in the next chapter.

## 6.1 OATS: Study of an Imagery Archive Browser

OATS<sup>2</sup> is a generic search and browse interface for large collections of geospatial imagery. It was built as a successor to the ERIC [73] system which managed an archive of GMS-5 imagery held by ACSys. OATS was widely demonstrated, including a high-profile presence at ATUG'97<sup>3</sup>, and attracted interest from research and military users. It is relevant to this thesis for two reasons: because it was designed to be independent of any particular type of geospatial data; and because it uses some similar performance techniques to RAPID. In particular it demonstrates the use and viability of the level of detail, scheduling and caching techniques described in Chapter 3. It was one of the first systems developed by the ACSys CRC to explore dissemination of geospatial imagery. As such the limitations of OATS are as informative as its strengths.

### 6.1.1 Overview and Requirements

OATS was an imagery dissemination system, pure and simple, with no thought of image processing. It provided a client-side searching, browsing and access interface to large archives of geospatial imagery. Figure 6.1 depicts the OATS client in use with the ACSys GMS-5 archive. The client provided a two-dimensional view of each dataset, with additional controls to navigate between channels and view ephemeral data and metadata.

The major requirement of OATS was that it should be able to support a very broad range of imagery types and archives. As discussed in Chapter 2, geospatial imagery from different sources can have very different characteristics, visible channels and associated metadata. OATS was not tailored to any one single type of data or archive: instead it provided a simple, generic framework for browsing and viewing many different types of imagery. Without compromising this generality, two assumptions were made about the imagery accessible to OATS:

1. that images would form time sequences with navigation in temporal order a common event; and
2. that images would be available at up to three levels of detail.

---

<sup>2</sup>OATS is an acronym for Online Archive Traversal System.

<sup>3</sup>The Australian Telecommunications Users Group Conference, the premier gathering of the telecommunications industry in Australia

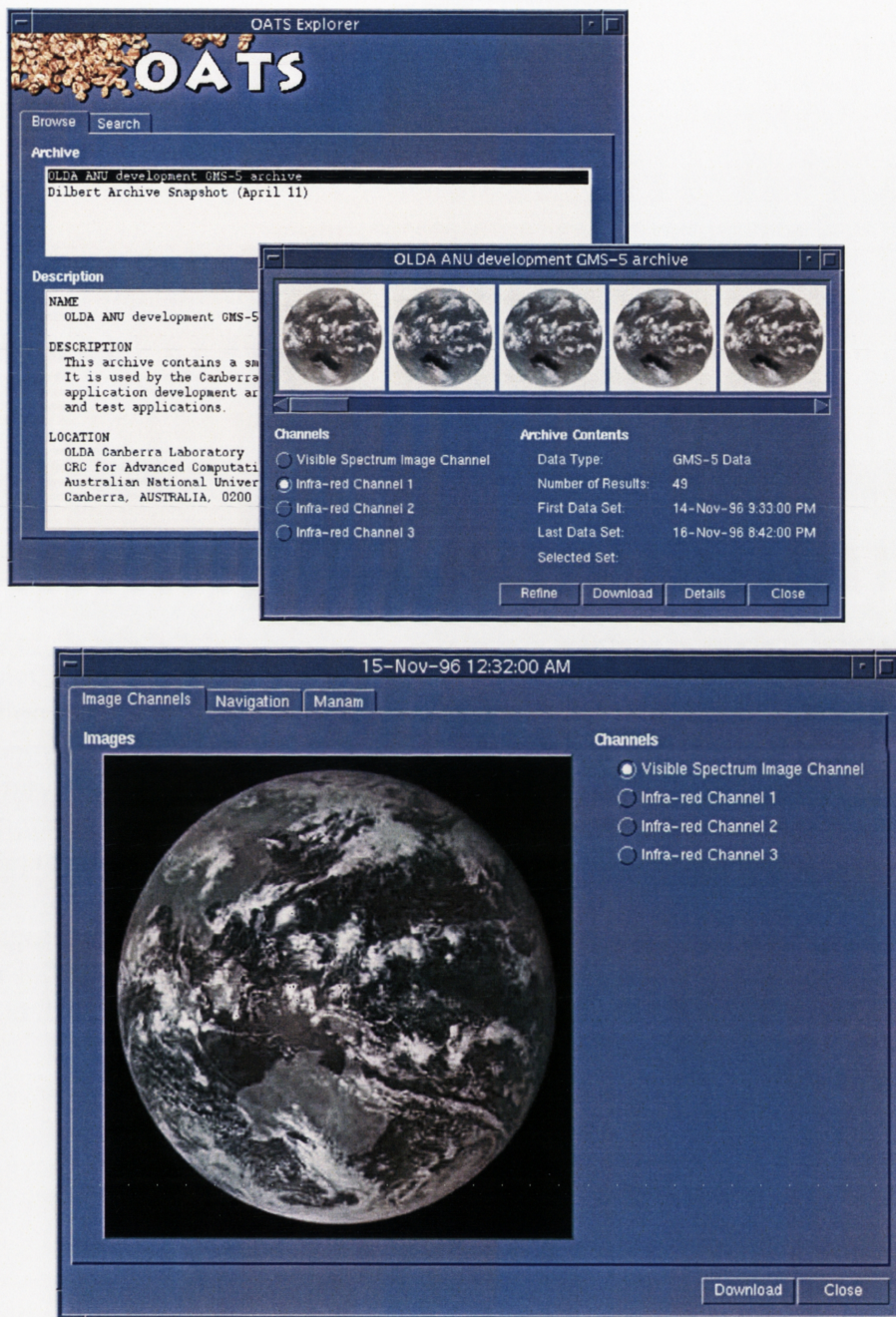


Figure 6.1: The OATS client exploring an archive of GMS-5 Images



The OATS interface was designed to allow efficient navigation in the temporal dimension. The GMS-5 satellite produces 28 sets of imagery every day, and the ACSys archive spanned several years. Other archives also contained data with significant variation over time. So the emphasis for OATS was on browsing time variant data, with temporal locality providing a useful heuristic for cache optimisation and speculative fetching.

OATS managed imagery at three different levels of detail. The lowest level consisted of small thumbnail images with a resolution in each dimension of a hundred pixels or so. Thumbnails allow a user to quickly scan and review the gross features of a dataset, without having to download large amounts of data. The second level of detail was designed to match the screen size of a typical user's workstation, with a resolution of 500 - 1000 pixels in each dimension. These medium resolution images were large enough to saturate the user's display device, without requiring the full dataset. Finally, the full imagery<sup>4</sup> was available when required. Although crude, this approach to level of detail was very sensible for a two-dimensional client interface. No tiling mechanism was associated with OATS imagery, but users could select subsets of an image.

### 6.1.2 Architecture and Implementation

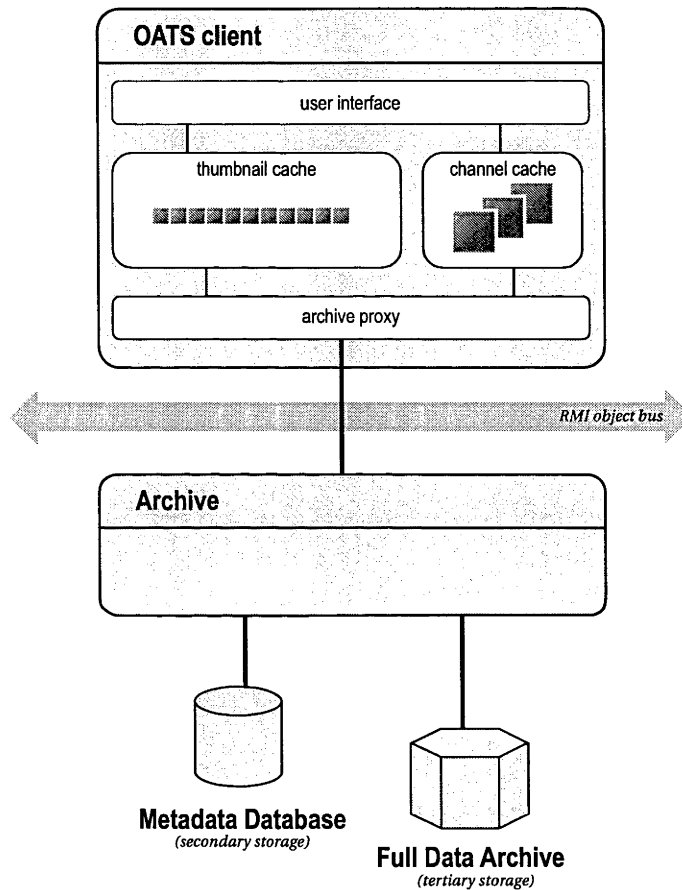
The architecture of OATS, depicted in Figure 6.2, bears some obvious relations to RAPID. It used a simple interface to an imagery archive that was easily implemented as a wrapper through the Adaptor pattern. It was independent of any specific type of data: an abstract model of type was used to describe the structure and components of an image, and the system adapted itself based on that description. This is very similar to the descriptive data passed between RAPID operators in a `Connection` object. A simple level of detail mechanism was employed to manage data size. OATS was not designed to perform any image processing so the approach to disseminating data was very simple: data was distributed directly from archives to clients across an object bus. Finally, caching strategies were used along with speculative data fetching to improve responsiveness, and to optimise temporal navigation.

OATS and the GMS-5 archive were implemented in three parts: a browser client, a metadata archive manager and a large data repository kept on a tertiary storage system. The browser client was implemented in 3500 lines of Java code with a clear separation of data handling, archive interaction and user interface code. The full GMS-5 archive was stored on a migrating tape file system using HDF [33] files<sup>5</sup>. To guarantee good interactive performance a separate metadata catalogue was maintained on a disk array to support fast searching and browsing of thumbnail images.

---

<sup>4</sup>Which for the GMS-5 archive was 4 channels of 2291x2291 pixels, with 3 IR channels stored at 8 bits per pixels and one visible channel at 6 bits per pixel.

<sup>5</sup>HDF is the Hierarchical Data Format developed by the NCSA and widely used as a general purpose file format for large scientific datasets and imagery.



**Figure 6.2:** The architecture of OATS. The client application cached images at two different levels of detail with speculative fetch policies used to improve cache hit rates. Interactions with remote archives took place through an object bus. The GMS-5 archive kept a database of imagery metadata on high-speed disk array, and the full data archive on a massive tertiary storage facility.

### 6.1.3 Analysis

There are three interesting lessons to draw from the OATS project. First, the approach to data-type independence directly informed the design of the `Connection` object in RAPID. Although its primary use was to serve an archive of GMS-5 imagery it was trivial to modify OATS to support other types of imagery (including an archive of the “Dilbert Zone” comic strip!). Despite radical differences in image formats, channel structure and archive interfaces, no code changes were required to support the basic data handling or movement associated with the new type. The ease with which OATS supported new types demonstrates the generality of the `Connection` object included in RAPID.

The second lesson to draw from OATS relates to its use of caching and speculative fetching to improve responsiveness. Even though it was a two-dimensional browse client, with limited responsiveness requirements, it used caching and speculative fetching to improve performance of temporal navigation. Even when the client and archive were situated on the same local area network, performance of the client was poor unless caching was used. A speculative fetch policy was also used to retrieve thumbnails and channels from medium-resolution images. These policies were designed to optimise the hit rate of the client image cache when a user iterated through a set of images in temporal order. That such mechanisms were required for a near-time browser emphasise their importance for real-time visualisation.

The third useful lesson from OATS relates to throughput problems when accessing remote archives. The core `Archive` interface, used both to search for images and to access them, was implemented in Java RMI. Transfer of bulk image data occurred over the RMI object bus. Unfortunately, like many distributed object technologies, RMI is not optimised for bulk data transfer. Consequently throughput became a major limit to the performance of OATS archives when viewing medium and full resolution images. This highlights the need to construct pipelines from efficient bulk data transfer mechanisms.

### 6.1.4 Summary of OATS

- Provided searching and browsing access to geospatial image repositories.
- Supported many different types of imagery.
- Used caching and speculative fetching to improve the performance of temporal navigation.
- Supported a crude level-of-detail mechanism.
- Poor throughput due to use of ORB-style communication mechanisms.

## 6.2 IMAD: Pipelined Imagery Dissemination

The Imagery Management and Dissemination (IMAD) Project [48] is run by the Australian Defence Science and Technology Organisation (DSTO). Its brief is to develop a national system to manage and access the many different types of imagery used

---

by Australia's armed forces. In early 1997 members of the IMAD team reviewed the work of the ACSys OLDA program, particularly the Comet and OATS projects. The review revealed a high degree of commonality between the two research groups, and was the catalyst for a fruitful collaboration. This case study is one of the results of that collaboration. It also documents the second in the family of dissemination systems designed by ACSys. IMAD is an ongoing project. This study covers that period of the IMAD project when the author was closely involved with it, and briefly reviews subsequent developments.

### 6.2.1 Overview and Requirements

Geospatial imagery is particularly significant for military command, control, communications and intelligence (C3I) applications. It is acquired from a range of different sources and held in facilities around the country. The challenge for the IMAD project is to make all this disparate and decentralised imagery available to C3I applications running wherever members of the Australian defence forces are deployed.

This task is complicated by three main requirements: to maximise the use of civilian technology (often referred to as COTS<sup>6</sup> technology); to support a range of different client applications and networks with greatly varied performance characteristics; and to support on-demand image processing as part of the dissemination process. The use of COTS technology focused on two standards: imagery should be held in a GIAS-compliant repository [149]; and the whole dissemination system be implemented as a set of reusable CORBA [105] services.

Clients of IMAD may vary greatly in terms of both visualisation capabilities and network bandwidth. The imagery used by photographic analysts with high performance workstations and broadband networks, may also be used by commanders in the field working with portable hardware and very low bandwidth network connections. Support for this later class of user is a very high priority and implies use of image compression and other techniques to optimise throughput. Managing the diversity of clients also presents an application management problem.

Military users have a variety of image processing needs. Automatic feature detection algorithms are an area of active research, and have the potential to revolutionise the use of geospatial imagery. Such algorithms are very demanding and require high-performance computing facilities. While these techniques evolve, feature detection is still performed manually by highly skilled analysts. Over time analysts build up a collection of valuable annotations for each image. Fusing annotations and other data sets with imagery is, in itself, an important requirement.

The work of the IMAD project has great relevance to the RAPID architecture. In most respects the IMAD applications are ideal candidates for use of RAPID. While collaborative data sharing is not a high priority, throughput and, to a lesser extent, responsiveness are crucial. The problems of application management are also the subject of ongoing research at DSTO.

---

<sup>6</sup>Commercial Off The Shelf

**Machines**

Dell OptiPlex GL+ 5100 desktop PC – 100MHz Pentium processor, 32MB RAM, Windows 95

Sun SparcStation 10 – dual 40MHz SuperSparc processors, 164MB RAM, Solaris 2.5

Silicon Graphics 02 – 180MHz MIPS R5000 processor, 64MB RAM, Irix 6.3

Sun Enterprise E3000 – dual 250MHz UltraSparc processors, 512MB RAM, Solaris 2.5.1

Digital AlphaStation 600 – 266MHz Alpha EV5 processor, 128MB RAM, Digital Unix 4.0c

**Networks**

10 mbps switched Ethernet

100 mbps switched Fast Ethernet

155 mbps ATM (Digital Gigaswitch)

**Software**

Java Development Kit (JDK) 1.1

JVMs with support for interpreted execution, JIT compilation and native code translation

Visigentic VisiBroker 3.0

---

**Table 6.1:** The test environment used for IMAD experimental work

## 6.2.2 Experimental Analysis

Because of the throughput problems experienced with OATS, a number of performance experiments were conducted for IMAD. These sought to evaluate tiling issues and measure the performance of COTS object middleware for bulk data transfer [140]. The results of these experiments also motivated the design of RAPID. Complete details of the experimental work undertaken by the author have already been published [140]. Briefly, the experiments sought to quantify two issues:

1. the impact of tile size on communication throughput, and
2. the performance of a commercial CORBA implementation (COTS technology) compared to that of low-level TCP sockets

The first issue is very significant to RAPID since it does not force a common tiling policy be used between operators, or mandate a minimum tile size. The second issue is equally significant since it motivates the use of separate service negotiation and service provision interfaces. A third issue that came to light during the experimental work was the need to short circuit network-oriented communications when two operators are located on the same host.

Experiments were run on a range of machine architectures and networks ranging from commodity PCs and a 10mbps Ethernet to large symmetric multi-processors and 155mbps ATM networks. Table 6.1 summarises the significant details of the test environment.

The experiments consisted of joining two null operators and measuring the length of time required to transfer a large image from one to the other. Various test images were used, typically several hundred megabytes in size. The test images were

transferred as a collection of tiles, with only one tile request outstanding at any time. Three different transfer mechanisms were available to the operators: a bulk data transfer bus using native TCP sockets; and two high-level object buses using the Visigenic VisiBroker ORB [151] and Java RMI<sup>7</sup>.

### 6.2.2.1 Communications Throughput

Figure 6.3 presents the throughput results measured on each of the three networks for socket and ORB transfer mechanisms. Several things are notable about these results. First, performance of the ORB is very sensitive to the size of tiles used: peak performance requires tiles of at least 256K. Note, however, that in many environments the ORB was unable to deal with very large tile sizes and simply crashed the application. Fine-tuning an ORB for peak throughput is a manually intensive process. It also places considerable restrictions on the higher level imagery applications, by requiring that they adapt the size of their tile requests to optimise throughput. By contrast the performance of the socket was far less sensitive to tile size.

A second notable feature of the experimental results is that, while the ORB performs relatively well over slow networks, it falls away dramatically over high-speed networks. Figure 6.3(i) shows that the peak ORB throughput over a 10mbps Ethernet is 82% of the peak socket performance. However, over a 100mbps Ethernet – 6.3(ii) – the ORB can only manage 39% of the socket throughput. For a 155mbps ATM network – 6.3(iii) – the ORB manages only 32% of the socket rate. These results reinforce a widely reported phenomenon: that ORB performance is not optimised to high-speed bulk data transfers [42, 43]. An interesting aside was that the socket implementation was able to completely saturate the ATM network, even with interpreted versions of the Java virtual machine<sup>8</sup>, dispelling the myth that Java is a performance bottleneck for network applications.

Finally the results in Figure 6.3(iii) demonstrate how greatly the use of the Nagle algorithm [131, pages 202–204] can affect ORB performance<sup>9</sup>. The Nagle algorithm is used to aggregate messages within a TCP stack to prevent flooding a wide area network with small packets. It does this by imposing a small delay on the transmission of individual packets. This is known to cause problems for certain patterns of traffic, and greatly affects the performance of the ORB. For the ATM test network the use of the Nagle algorithm causes highly erratic behaviour for the ORB with respect to tile size: in some cases throughput is less than a hundred bytes per second.

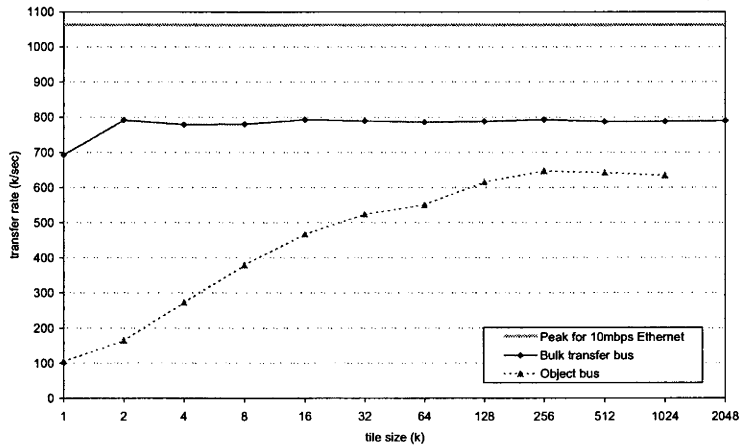
These results clearly illustrate the need for a specialised bulk data transfer mechanism to complement the use of a high level object bus. This separation is cleanly made in RAPID though the use of the `Operator` interface for performance insensitive

---

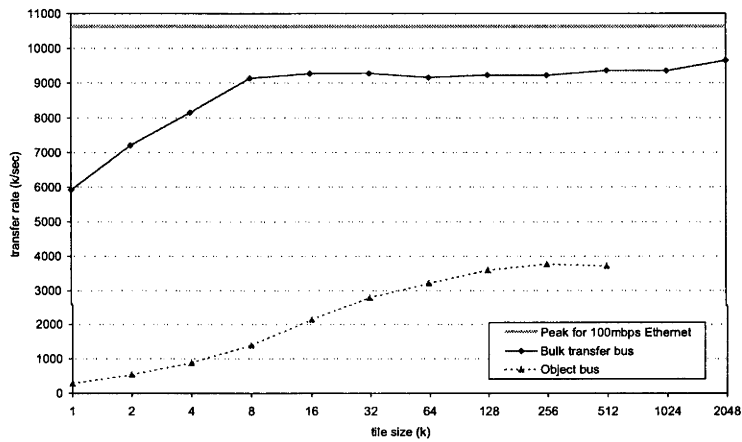
<sup>7</sup>Results for the RMI implementation are not presented here since the VisiBroker implementation consistently outperformed it.

<sup>8</sup>The interpreted socket implementation consistently achieved throughput rates equivalent to 96% of the theoretical peak rate for IP over ATM.

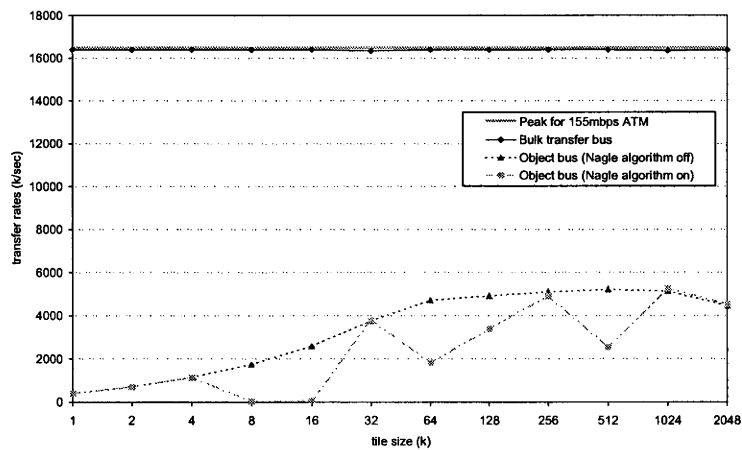
<sup>9</sup>Disabling the Nagle algorithm is often referred to as use of the “no delay” option for a TCP socket. Not all TCP implementations allow the Nagle algorithm to be turned on or off for individual sockets.



(i) 10mbps Ethernet



(ii) 100mbps Ethernet



(ii) 155mbps ATM

**Figure 6.3:** IMAD throughput measurements for three different networks. The performance of an ORB (Object bus) is compared with that of a raw TCP socket (bulk transfer bus). Note the difference in scales on the vertical axes of each graph Use of the Nagle algorithm can cause ORB performance to be highly erratic over high-bandwidth connections.

---

functionality, and the lightweight RAPPORT protocol over low-level optimised Links. Within the GIAS standard there is no provision for bulk data transfer mechanisms: yet such are clearly required to achieve high rates of throughput.

### 6.2.2.2 Efficiency of Loop-back Communications

An unexpected result came from testing loop-back communication performance. To measure the peak throughput of various TCP stacks the tests were performed with both operators running on the same machine. This provides a measure of the loop-back performance of each machine and gives an indication of performance in an idealised network. More usefully, it measures the effectiveness of the socket and the ORB as inter-process communication (IPC) mechanisms. This is particularly relevant to RAPID in situations when two operators are located on the same physical machine, such as a high performance server.

The results of loop-back tests for the two highest performing machines are presented in Figure 6.4. In both cases the socket implementation comprehensively outperforms the ORB. More importantly, Figure 6.4 reveals how completely inappropriate network-oriented primitives are for inter-process communications. Both machines have processor memory-bandwidth of several gigabytes per second, yet can only transfer data between processes at a rate of tens of megabytes per second. Obviously a more efficient IPC primitive, such as shared memory, should be used for bulk data transfer between operators running on the same machine.

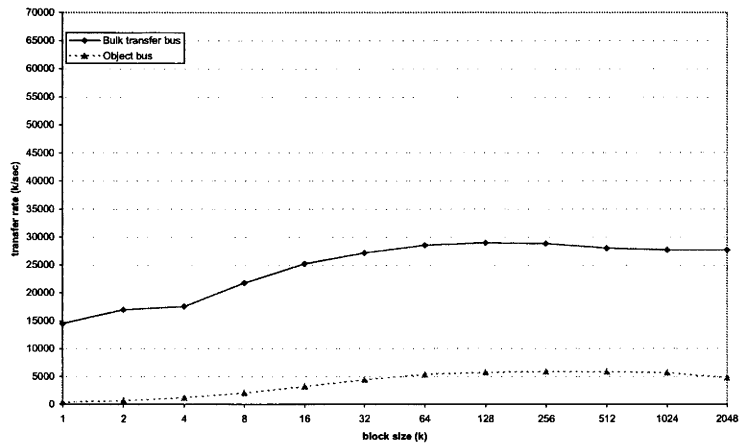
The RAPID `Connection` and `Link` objects allow such mechanisms to be used transparently.

## 6.2.3 Architecture and Relationship to RAPID

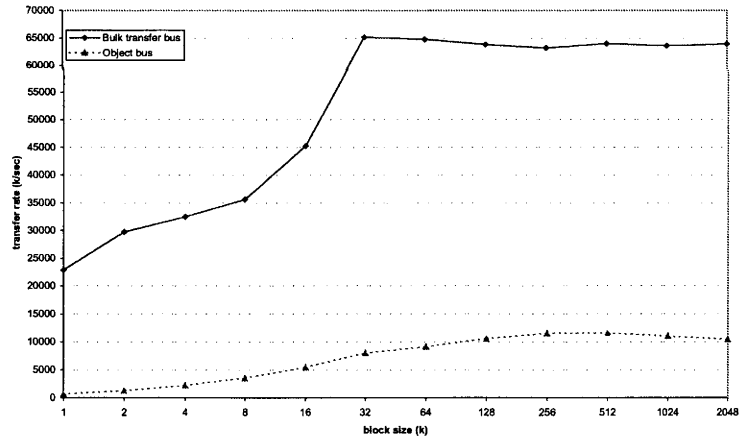
The results of experimental work led to the development of a streaming, pipelined architecture for imagery dissemination, known as IMAD-1. This was a subset of RAPID without the focus on responsiveness or collaborative data sharing and limited support for application management. The main aim of the design was to optimise throughput of two COTS technologies: CORBA and GIAS. As with RAPID, the solution was predicated around a two-phase model of service, with non-critical negotiation performed through a CORBA object bus and time-critical data delivery provided through an optimised operator network.

Four general categories of actor formed the basis of IMAD-1: service traders; service managers; operator factories and operators. These four actors are also present in RAPID. Figure 6.5 describes the four base actors and depicts their interactions. From these generic actors a collection of specific services was proposed for IMAD-1, including:



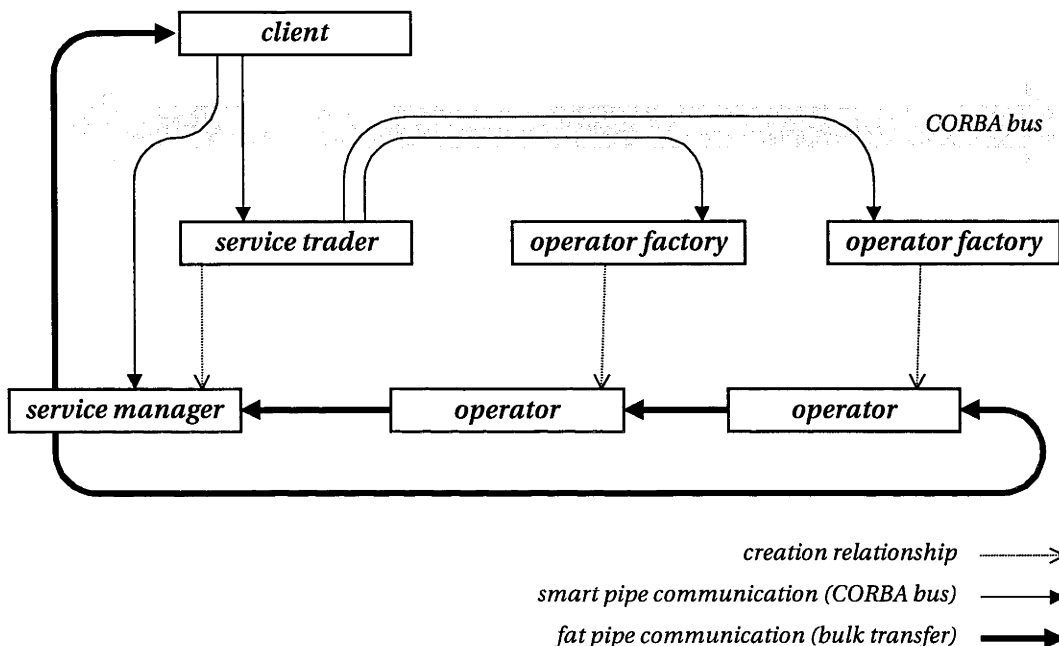


(i) Digital Alphastation 600



(ii) Sun Enterprise E3000

Figure 6.4: Throughput measured over the loopback interface for two high performance machines.



**service trader** – the first point of contact for client applications, used by the client to find or construct a service manager which it uses for operational requests. A trader represents a class of service available to clients. The trader negotiates with the client to determine its requirements, and then directs the client to a service manager which can satisfy the operational requests. The manager may be created by the trader specifically for the client, or may be drawn from a pool of idle managers.

**service manager** – built by a trader for use by a client when it is in the operational stage. A manager represents an instance of a service. Each manager is unique to a single client for the lifetime of that client’s operational requirements. In general service managers use a network of operators to satisfy client requests.

**operator factory** – produces the operators used by a service trader to build a service manager. Clients interact indirectly with operator factories, through a service trader.

**operator** – a basic operation of a computation. Well-defined input and output interfaces allow operators to be tied together to form networks. Clients interact indirectly with operators, through a service manager.

**Figure 6.5:** Base actors in the IMAD-1 architecture: their roles, interactions and communication patterns.

**Tiles Trader** – A service trader that allows clients to search image archives and access images as a collection of compressed tiles. Essentially a `PipelineTrader`.

**Tiles Manager** – The service manager created by the tile trader, and used by clients to access tiles. Equivalent to the `PipelineManager`.

**Catalogue Service** – Used during service negotiation to search archives for appropriate imagery. No equivalent exists in RAPID since searching is considered to be a separate problem.

**Retrieval Factory** – A factory for operators used to retrieve data from an imagery archive. Identical to the `RAPID ImageryArchive`.

**Retrieval Operator** – An `ImageAccessor` operator for retrieving particular images from an archive.

**Compression Operator** – A `FilterFactory` for operators that compress image tiles.

**Compression Factory** – A `Filter` operator that compresses imagery tiles.

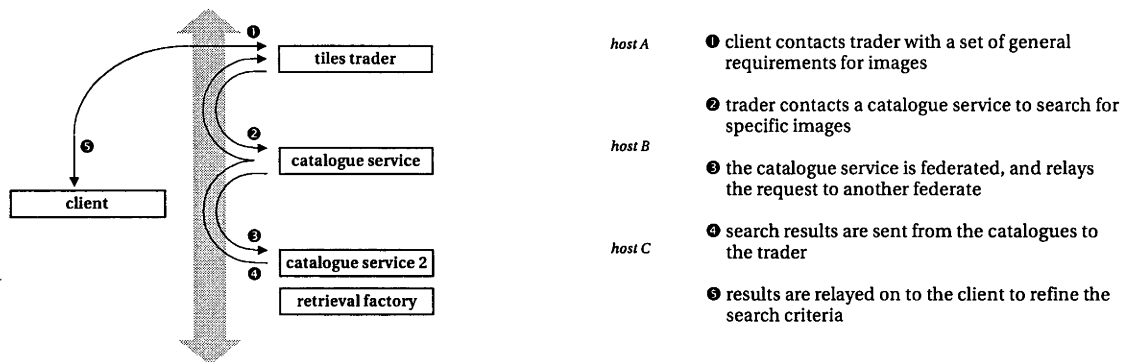
Figure 6.6 demonstrates how these actors are used for service negotiation, creation and provision. Once again the interactions between components are very similar to those in RAPID.

Despite these similarities, the IMAD-1 architecture was more limited in scope than RAPID. It provided only limited capabilities for application management, based on Traders and Factories. No global directory service was used, so all service advertisement and resource discovery was by way of Traders. Dissemination pipelines were strictly single-user entities: no consideration was given to collaborative issues and the associated requirement for concurrent access to the pipeline. Indeed the entire pipeline was more loosely defined: there were no standard interactions between operators and no standard protocol, such as RAPPOR. The single-user focus also affected the role of service managers. These were responsible not just for managing operator life-cycles and connecting client applications, but also for scheduling pipeline processing and interacting with clients directly. As such they performed the duties of both the `PipelineManager` and the RACE.

## 6.2.4 Analysis of the Architecture

The IMAD-1 architecture successfully meets its primary requirement of optimising throughput for dissemination pipelines based on COTS technology. However, subsequent analysis of the architecture reveals limitations in terms of application management, difficulties in the interactions between operators, and poor support for collaborative applications or end site caching.

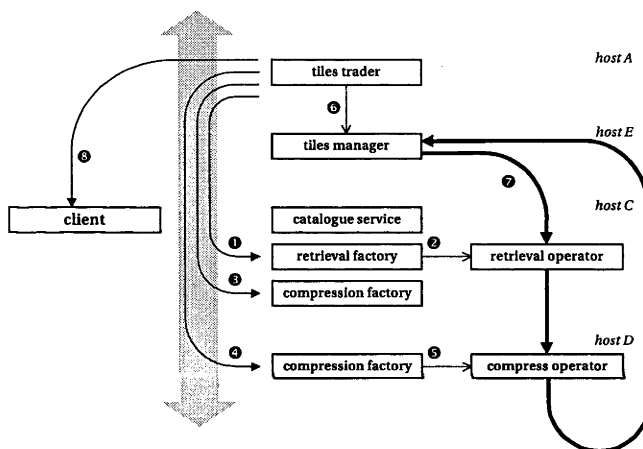
IMAD-1 did not consider the problems of distributed application management in great detail. The use of a two-phase service model allows resource reservation and management policies to be implemented by service factories. However, the lack of a standard namespace for service advertisements adds complexity to factories, traders and clients alike. Factories must track and register themselves with all traders who might use their services. Traders must record which factories are available and how operators can be combined. To support system-wide searches Traders must be



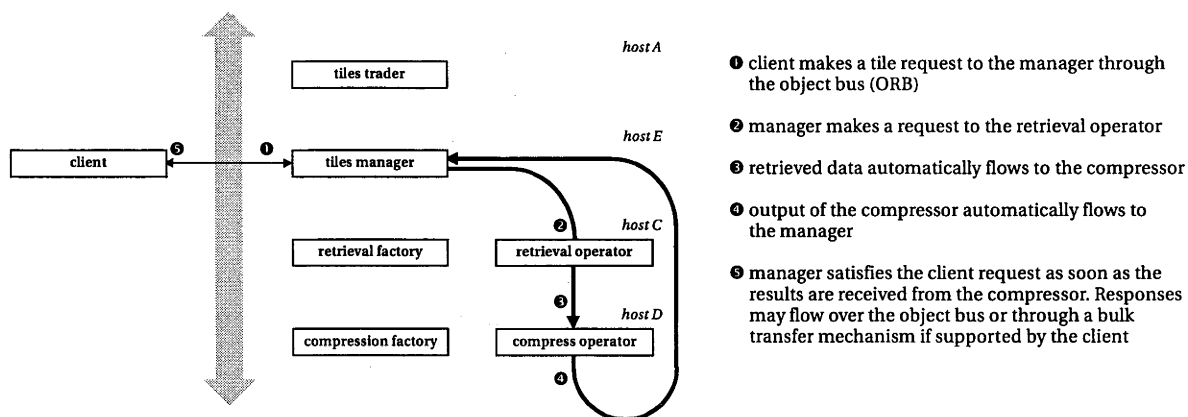
- host A ❶ client contacts trader with a set of general requirements for images
- host B ❷ trader contacts a catalogue service to search for specific images
- host B ❸ the catalogue service is federated, and relays the request to another federate
- host C ❹ search results are sent from the catalogues to the trader
- host C ❺ results are relayed on to the client to refine the search criteria

(i) Service negotiation

- ❶ trader contacts the archive retrieval factory
- ❷ factory produces a retrieval operator for images
- ❸ trader contacts compression factory associated with the archive, but it doesn't support the client's desired compression algorithm
- ❹ trader finds a second compression service which does support the required algorithm
- ❺ factory produces a compression operator
- ❻ trader produces a manager for the pipeline and connects the operators
- ❼ manager initialises the pipeline and all operators
- ❽ trader returns the manager to the client



(ii) Service initialisation



- ❶ client makes a tile request to the manager through the object bus (ORB)
- ❷ manager makes a request to the retrieval operator
- ❸ retrieved data automatically flows to the compressor
- ❹ output of the compressor automatically flows to the manager
- ❺ manager satisfies the client request as soon as the results are received from the compressor. Responses may flow over the object bus or through a bulk transfer mechanism if supported by the client

(iii) Service provision

**Figure 6.6:** Interactions between IMAD-1 actors during (i) negotiation, (ii) initialisation, and (iii) provision of services. These interactions are very similar to those in RAPID, with the one notable difference in request propagation due to the informality in request and response flow semantics.

federated. Clients have no standard location in which to find services and consequently must solve the familiar distributed system boot-strap problem: how to get the first pointer to the first trader? Finally, there is no location information from which to make pipeline routing and construction decisions. All of these problems can be solved by computational grids with an appropriate resource schema.

No standard set of interactions was defined for operators in an IMAD dissemination pipeline. Data flows can be initiated from either end of a connection: there are no standard request, response or flow control semantics. This ambiguity is deliberate: an operator can be implemented with a very thin wrapper (Facade) around an existing image processing system. However, it also adds to the complexity of the Trader services, and in some cases limits the performance of the pipeline. Since no standard protocol or set of interactions are specified for the pipeline, great care is required when connecting operators. Traders are responsible for choosing operators that are known to work together. So in addition to recording what services are available, a Trader has to determine what services can be used together. The non-standard interactions in the pipeline also make it difficult for a Manager to schedule or optimise the flow of data. Finally, operators have to buffer an unknown amount of data. With no flow control mechanism in the pipeline it falls to the Trader to ensure that operators were not connected in such a way that one could overwhelm another. These limitations are all addressed by the RAPPORT protocol.

The IMAD-1 architecture was not really designed for collaborative or multi-user applications, so concurrent access to the pipeline was simply not considered. All client interactions take place through a single service Manager. It would certainly be possible to build a manager that supported multiple simultaneous clients and serialised concurrent requests. However, for large numbers of clients the manager would become an obvious bottleneck. More significantly use of a single, centralised manager doesn't allow for any local caching of results at client sites. It also requires that all clients access the same data, with no obvious way of supporting clients with different visualisation requirements. This is because the IMAD-1 manager has to fulfil the roles of both `PipelineManager` who controls and moderates access to a service, and `RACE` which acts as client application interface and cache. To aid collaborative systems, `RAPID` decouples these two roles, and also allows concurrent access to the pipeline.

### 6.2.5 Subsequent Work

Considerable development has taken place in the three years since the IMAD-1 architecture was proposed. In collaboration with the ACSys OLDA team, a GIAS compliant image repository has been developed [20]. This repository was used by OLDA to explore the potential for active archives, which fuse data processing and data warehousing functions [56]. It was used by IMAD as the foundation for a range of imagery dissemination services [48]. A file-oriented dissemination service was built for non-interactive applications. Progressive download and near-time interactive dissemination services were developed to support two dimensional visualisation

---

clients without real-time requirements. Finally an intelligent dissemination service was developed which speculatively distributes imagery using some of the scheduling policies presented in section 3.1.2. However, this service was not designed for real-time clients.

More recently, the IMAD group have developed a generalised pipeline architecture, known as iFlow [93], which supports many different types of streamed data. This architecture is designed for general use in C3I applications, and attests to the success the group have had with pipelined imagery dissemination.

### 6.2.6 Summary of IMAD

- Experimental work revealed throughput problems inherent with ORB-style communications.
- A subset of RAPID, known as IMAD-1, was proposed to support dissemination pipelines with efficient bulk data transfer.
- Application Management was only considered in a limited way. A large-scale deployment of IMAD-1 would present considerable resource discovery and management problems.
- Operator interactions, request handling semantics and flow control were not specified in detail.
- No support was provided for collaborative applications or concurrent access to the pipeline.
- The roles of PipelineManager and RACE were combined in IMAD, which precludes use of site tile caches described in Section 3.1.3 and limits responsiveness.
- Searching functions were integrated into IMAD-1 but are difficult to standardise across different applications. Consequently RAPID focuses solely on the dissemination problem.
- The ongoing work of the IMAD project demonstrates how a pipelined model of imagery dissemination can be very effective and can achieve high rates of throughput.

## 6.3 CROP: Optimising Throughput

The third and final dissemination system presented in this chapter was the result of the ACSys CROP [61] project. Developed in early 1998, the CROP project refined the dissemination pipeline model and introduced an elegant way of abstracting the underlying communications mechanisms used in a pipeline. The design of CROP was greatly influenced by the experimental work presented in section 6.2.2 and the IMAD-1 architecture. In practice CROP achieved excellent rates of throughput and is indicative of the performance possible with RAPID. It is presented here as a final example of how processing pipelines can be used for imagery dissemination, and to

The screenshot shows a web-based interface for the CROP client application, divided into three main sections:

- 1. Select region of interest (rubberband):** Features a map of Australia with state/territory labels (Northern Territory, Queensland, Western Australia, South Australia, New South Wales, Victoria, Tasmania). Below the map, it displays 'AREA SELECTED' with coordinates: 'From: 5.5 S lat 110.5 E long (NW corner)' and 'To: 46.5 S lat 156.5 E long (SE corner)'. A button 'Set region to area selected' is present.
- 2. Select time series:** Includes a 'Day' calendar grid (1-31), 'Month' and 'Year' dropdown menus, and a grid of year-month combinations (e.g., 1989-1998, Jan-Mar, Apr-Jun, Jul-Sep, Oct-Dec). Buttons for 'Add date', 'Remove date', and 'Reset series' are located below the calendar.
- 3. Process data:** Contains buttons for 'Compute dataset statistics', 'Interrupt computation', and 'Show results'. A 'Status:' field is at the bottom.

**Figure 6.7:** A sample CROP client application which provides a user interface to one particular form of processing pipeline

underline the techniques needed to optimise throughput. It also highlights the need for effective application management techniques.

### 6.3.1 Overview and Requirements

The CROP project [61] was a joint undertaking between the ACSys CRC and Agrecon Pty Ltd, a private company which specialises in the use of GIS data and remote sensing systems [2]. The aim of the project was to provide simple and efficient access to earth observation data in a way that allowed domain specialists, such as Agrecon, to produce value-added imagery products. A prototype was developed to enable monitoring of crop growth and yield estimation across the country, for the benefit of clients such as the Australian Wheat Board. Figure 6.7 shows a CROP client application through which users interacted with the prototype.

Like IMAD, the CROP project was constrained in its use of technology. The main requirement for CROP was that it should integrate a collection of legacy image processing applications developed by Agrecon to perform tasks such as crop yield analysis. In some cases this legacy code was impossible to modify. In all cases it was designed for single machine processing, with no concept of distributed computation. The CROP system was essentially a runtime harness for running legacy code and providing transparent distribution of data.

The other significant requirement of CROP was that it should operate efficiently and at high rates of throughput. It was designed for use in cluster computing envi-

---

ronments, but also had significant support for wide area distribution. Unlike OATS and IMAD there was no requirement for interactivity in the CROP system, so issues of responsiveness and collaborative data sharing were not considered. However, application management was identified as a problem, and resource use and scheduling issues were considered.

### 6.3.2 Architecture

Like IMAD-1 and RAPID, the architecture of CROP centres on a dissemination pipeline. CROP performs two main tasks: scheduling and running processes; and automatically transferring data between hosts. Pipelines are formed between CROP services (operators), where each service encapsulates a single legacy processing element. Efficient movement of data between services takes place automatically and is completely transparent to the legacy code, thanks to an optimised *DataTransfer* facility. CROP also manages the cleanup and disposal of the pipeline after it has been used. The elements of CROP's architecture include:

**Registry** – CROP uses a two-level name service for registering resources. All machines which participate in a CROP system run a local registry which records services running on the machine. In addition there is a global registry at a well-known address, which records all machines in the system. The namespace is implemented in very simple terms: it is non-hierarchical and not replicated, it supports only limited searching and has no authentication or security mechanisms. No schema is maintained, but services can associate some metadata along with a global pointer.

**BasicService** – All CROP services are derived from an abstract *BasicService*, which introduces a high degree of observability into the system. It automatically handles registration in the CROP namespace. More usefully it provides an asynchronous event delivery system that allows clients to observe the state of any running service. An event filtering model allows clients to upload a simple filter to the remote service to limit the transmission of events.

**DataTransfer** – Data must be transferred between CROP machines transparently. The *DataTransfer* service is used to move *DataObjects* from one machine to another. A *DataObject* represents the output of a process/operation and encapsulates not just the data, but also the mechanism used to transfer the data. Its role is identical to that of the *Connection* object in RAPID. When a downstream operator wants to retrieve data from an upstream operator it retrieves a *DataObject*. This object automatically initiates the transfer through an appropriate mechanism: TCP sockets, ATM circuits or an IPC construct. It is responsible for buffering the data and cleaning up after the transfer has been completed.

**ProcessService** – Machines that have spare computational capacity run a *ProcessService* to execute processes on behalf of a client application. This is equivalent to *FilterFactory* used in RAPID.



**ScheduleService** – To coordinate the ProcessServices running on multiple machines CROP uses a simple ScheduleService. This manages a collection of machines and creates new processes on the most under-utilised machine. It uses a very simple scheduling heuristic, but still provides a useful degree of load-balance across a cluster.

**DataStore** – The access interface of an imagery archive is known as a DataStore, which is equivalent to the ImageAccessor class in RAPID. Clients use the CatalogService to identify appropriate images, then retrieve them as DataObjects from a DataStore.

**CROPServer** – The prototype CROP application involved an extensive sequence of processing operations, arranged in a long pipeline. A CROPServer was developed to automate the construction of these pipelines. This is equivalent to the function of a PipelineTrader in RAPID.

Aside from naming issues, the significant architectural differences between CROP and IMAD-1 are in the roles of the DataTransfer service and the Registry. Where IMAD-1 does not include a well-defined set of interactions between operators CROP does. A DataObject describes not only the application data that will flow down the pipeline, but also the communication mechanism that should be used to form one stage of the pipeline. DataObjects allow the behaviour of an operator to be extended dynamically to accommodate new protocols or communications mechanisms. The CROP Registry also provides a partial solution to the service advertisement and discovery problems encountered in IMAD-1. CROP avoids the extensive use of Traders by providing a global namespace. Trader-style services still exist, but are much simpler to implement and are focused on supporting a specific class of application.

### 6.3.3 Analysis

There are three interesting conclusions to draw from CROP: the high rates of throughput performance it achieves; the use of an abstract mechanism for connecting operators; and the use of a flat namespace for application management. Each of these conclusions has direct relevance to RAPID.

Perhaps the most important characteristic of CROP is the very high rates of data throughput it achieved. The DataTransfer service consistently performed at a rate of more than 16 megabytes per second over a 155mbps ATM network – effectively saturating the network<sup>10</sup>. In practice, none of the processing operations could match this rate of throughput. Hence CROP achieved the goal of any dissemination pipeline, which is to ensure that dissemination rates are bound by computation not communication.

One of the more elegant ideas in CROP was the use of an abstract connection mechanism. The flexibility and transparency of data movement was due to the use

---

<sup>10</sup>Extensive performance measurements are not presented for CROP, since there was little variation in performance within the operational environment. Further details are available in the previously published material [61].

---

of `DataObjects` to connect operators in the pipeline. By encapsulating the choice of link mechanism within these objects, CROP services could vary communication primitives on a connection-by-connection basis. This also allowed for dynamic extensibility of operators to support new communications mechanisms. Although the same flexibility was possible in IMAD-1, it had to be implemented on an ad-hoc basis and required additional intelligence in the Trader and Manager actors. The simplicity of the CROP approach is compelling and contributed to the use of `Connection` objects in RAPID.

The final conclusion to draw from CROP relates to its use of a simple registry for service advertisement and other application management. Although adequate for a cluster-area system, the CROP registry service would not suite large scale deployment in a wide-area network. The flat namespace and minimal schema are very limiting. Use of a home-grown, application-specific directory service makes it hard to integrate new components and precludes the use of standard management tools. Finally, the approach used to scheduling was extremely simplistic. Considerably more effective scheduling systems have been developed for metacomputing environments and computational grids. Consequently, although CROP was an improvement over IMAD-1 in terms of application management, it was still difficult to build and run large pipelines.

#### 6.3.4 Summary of CROP

- Refined the pipeline model developed for IMAD-1.
- Successfully provided a harness for running legacy image processing operations with transparent distribution of data between processes.
- Achieved excellent rates of throughput and consistently saturated a cluster-area network.
- Used an abstract connection mechanism that allowed great flexibility in the delivery of data downstream.
- Application management was considered but no comprehensive answers were developed. The limitations of application management motivated subsequent work with computational grids.

## 6.4 Summary

This chapter has reviewed three imagery dissemination systems, with a particular emphasis on the use of pipelining to achieve high rates of throughput. Most geospatial imagery archives provide searching interfaces through distributed object middleware, such as an ORB. The poor performance of ORBs can make them a major bottleneck when disseminating large datasets. The systems presented in this chapter have demonstrated how to overcome this bottleneck through the use of bulk transfer mechanisms, connected together to form dissemination pipelines. Such

pipelines can achieve very high rates of throughput, and also provide a framework for integrating legacy image processing applications.

Yet dissemination pipelines are not without limitations. Caching and speculative fetch policies are required to protect highly responsive clients from pipeline latency. More significantly, creating and using long pipelines brings with it a range of application management problems. Although both IMAD-1 and CROP projects considered this problem in part, both suffered for lack of a comprehensive management framework such as that defined for RAPID.

The systems presented in this chapter demonstrate many of the concepts in the RAPID architecture. They demonstrate how pipelines can be formed and used effectively, and the value in an abstract connection mechanism. They also provide the motivation for considering application management in more detail. This is the subject of the next chapter, which provides a final case study into an management system known as the vGrid.

---

# vGrid: a Study in Application Management

---

The previous chapters focused on runtime performance issues in detail, with a particular emphasis on responsiveness and throughput. We will now consider *application management* through a case study of the vGrid, a metacomputing system designed to support virtual environments. The vGrid was developed in early 1999 following experiences with Comet and the CEOS demonstration, and with the i-Grid demonstrations at Supercomputing '98. It is a data-centric grid with particular support for the unique properties of VEs. It is also a superset of the application management framework included in RAPID, and so is a valuable proof of concept.

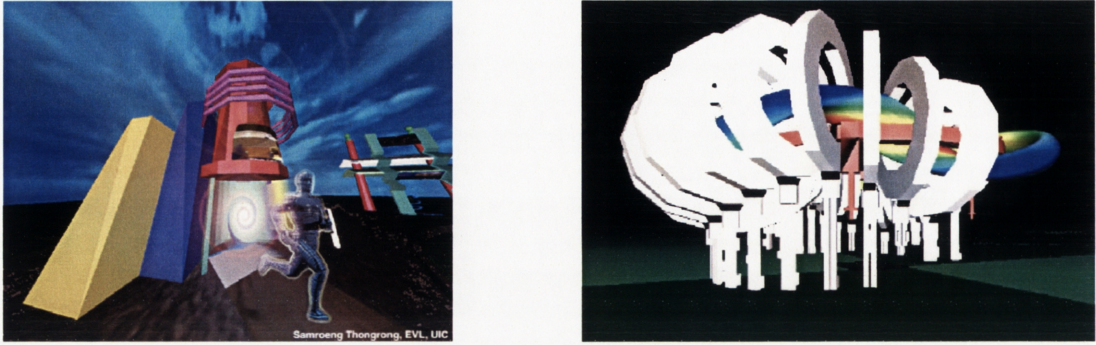
The structure of the chapter follows that of the Comet case study. Section 7.1 provides an overview of the vGrid and describes the motivation for its development. Section 7.2 clarifies the relationship between the vGrid and the RAPID application management framework. The architecture of the vGrid is very similar to that of RAPID, but with some minor differences to accommodate a broader class of application. These differences are reviewed in section 7.3. Evaluating a grid system is not simple, but section 7.4 provides an analysis of the vGrid and identifies strengths and weaknesses. Finally, section 7.5 provides a summary.

## 7.1 Overview of the vGrid

Before considering the design of the vGrid in detail, it is valuable to review the motivation for and goals of its development.

### 7.1.1 Motivation

The first motivation for the vGrid came from the CEOS demonstrations of imagery dissemination and processing. Although Comet required access to a range of storage, processing and network resources, all application management was performed by hand. Datasets were accessed, processing performed and results transferred through an ad-hoc collection of scripts and configuration files. Although the IMAD-1 and



**Figure 7.1:** The ACSys presence in the i-Grid demonstrations at Supercomputing '98 (left) consisted of a virtual world containing a model of plasma flows through the Heliac reactor (right).

CROP systems (discussed in the previous chapter) automated dissemination, they too suffered from application management problems.

A final motivation came from Supercomputing '98, where a large number of demonstrations were made under the auspices of the i-Grid [13] project. The i-Grid aimed to show how a computational grid could harness and exploit a collection of globally distributed resources. A central element of the i-Grid was a Collaborative Virtual Environment, based on Limbo/CAVERNsoft [85], which allowed participants around the globe to interact and visualise various results. The author was involved with this demonstration as a member of an Australian team based at the CRC for Advanced Computational Systems (ACSys). Figure 7.1 depicts the virtual world ACSys contributed to the i-Grid VE, demonstrating the plasma flows within the Heliac reactor.

While preparing for the i-Grid demonstration it became clear that the application management requirements of a CVE are quite different to those of other applications running in a computational grid. In particular, the i-Grid was a poor support vehicle for the Limbo VE because:

1. All configuration had to be performed manually. Whenever any change occurred in the application configuration, it had to be reflected to every participant and often involved manual intervention
2. The topology of connections between participants was far from optimal. A major feature of CAVERNsoft is the flexibility it affords applications in constructing efficient topologies for sharing data. Unfortunately the grid and application were not integrated closely enough to take advantage of this capability.
3. All users had to have the same set of executables to participate in the world. This restriction is quite reasonable for a concept demonstration, but in a production environment users are unlikely to have identical facilities or runtime systems.

4. The virtual worlds shown in the demonstration had been constructed specifically for the purposes of show. The contents of the world were carefully crafted months in advance to maximise the effect of the demonstration, rather than in response to any real user requirements. In a production environment worlds should be created dynamically as users determine a need.
5. A virtual lobby space was provided as a meeting place for collaborators and to provide portals between the major virtual worlds. This lobby was also statically created and designed only for the worlds run on i-Grid. A production environment should include a dynamic lobby service for matching collaborators with data sets and computational resources<sup>1</sup>.

The goal of the vGrid was to provide a data-centric grid to do a better job of managing visualisation systems and CVEs such as Comet and Limbo.

### 7.1.2 Requirements of Collaborative Virtual Environments

Virtual Environments are data-centric applications, which are very performance sensitive and have a number of soft-realtime requirements. Foremost among these is the need for low latency and high responsiveness. A grid can provide only limited support for the performance requirements of a VE, but can address other data-centric issues. Virtual Environments require support from a grid to perform the following tasks:

- getting at interesting data sets;
- transforming data to a visualisable form;
- sharing data through numerous different network protocols;
- constructing a topology of application-level connections;
- support for heterogeneity and diversity in: machine architecture, network bandwidth, rendering power, client software.

### 7.1.3 Goals

The general aim of the vGrid is to demonstrate how a data-centric grid can solve the application management problems of broadly distributed Virtual Environments. As such it provides answers to the standard resource management questions outlined in previous chapters. Beyond this, however, the vGrid seeks to overcome the specific limitations of the CEOS and i-Grid demonstrations by:

---

<sup>1</sup>This is one area where multiplayer video games, when considered as a form of Collaborative VE, offer considerable insight. Almost all multiplayer games now feature some form of lobby service to match players with game servers. Lobby services vary in complexity from simple chat spaces to sophisticated interest matching systems. However complex the service, it invariably becomes an essential part of the community that builds around a successful game. The i-Grid demonstration shows that, to be successful, Virtual Environments have no less a need for a lobby service or the sense of community it engenders.

- allowing users to advertise and find virtual worlds, and supporting dynamic lobby services;
- automatically configuring a client to join a virtual world;
- supporting interoperability between different clients;
- creating worlds dynamically for a user;
- allowing applications to optimise the topology of connections between peers;
- requiring minimal or no changes to client executables; and
- mandating no common network interface or protocol.

The last two points are particularly noteworthy. Though the vGrid provides many resources an advanced client could use, the basic intention is that the grid services should be provided transparently. Little or no code changes are required to integrate an existing VE into the vGrid. As a result, the vGrid does not require a standard approach to networking or a common application protocol. This was initially seen as a great strength of the approach, but also imposed some constraints which will be discussed shortly.

## **7.2 Relationship to RAPID**

The vGrid is a prototype for all the application management components of RAPID. It is actually a superset of the RAPID model, since it deals with a more general class of application and with arbitrary data structures. While RAPID is concerned exclusively with geospatial imagery, the vGrid is designed to support many different types of visualisation client and types of visual data.

The generality of the vGrid presents a broader set of application management issues than RAPID must consider. Since the vGrid is application-neutral it cannot define a standard network protocol, such as RAPPOR. Lacking a common protocol, the vGrid schema must model protocols as first class objects. It cannot rely on a common application topology, such as a dissemination pipeline: instead it must support a range of different topologies. It is not possible to define a single standard life-cycle for resource use.

By considering a more general set of issues, the vGrid demonstrates that the RAPID application management facilities are both practicable and effective.

## **7.3 Architecture**

The vGrid approach to application management is the same as that of RAPID. A resource schema published in a global directory service is used to describe the network environment and physical resources available for VEs. Virtual worlds – conceptually equivalent to pipelines – are a collection of different shared datasets. Each dataset is produced by a factory, which uses a negotiation interface to regulate use of resources. Traders are used to automate the construction of virtual worlds and isolate VE clients from the vGrid infrastructure.

---

But the added generality of the vGrid requires some extensions to the RAPID model. Foremost among these is the need to model data types and network protocols as first class objects. The vGrid also supports a much richer variety of Trader services than required for RAPID. Finally, there are subtle differences in the use of factories. vGrid supports two quite distinct types of factory: a *dataset factory* provides access to a collection of datasets and is conceptually similar to RAPID's Imagery-Archive; an *adaptor factory* produces *Adaptor* [40, pages 139–150] objects to bridge between protocols and help promote interoperability.

We will now review the architecture of the vGrid, as it differs from that of RAPID. Though the discussion that follows focuses on the differences between vGrid and RAPID, it should be noted that they are overwhelmingly similar. For an independent discussion of the vGrid the interested reader is referred to previously published work [139].

### 7.3.1 Resource Schema

Like RAPID, the vGrid defines a collection of standard metadata in a resource schema. This schema is published in a directory service and used by traders and clients to access resources and connect to virtual worlds. It is the glue that holds the various pieces of the grid together. Table 7.1 presents the vGrid resource schema in full. Much of the schema will look familiar after the discussion of Chapter 4. For example the attributes describing sites, machines and people are basically identical. The network model is simplified, and corresponds to the links described in RAPID's schema. The vGrid makes explicit the distinction between two different types of factory – Dataset factories and Adaptor factories – but these are otherwise similar to their RAPID equivalents.

Perhaps the most significant difference between the vGrid and RAPID schemas is in the treatment of virtual worlds. The concept of a *World* in the vGrid is equivalent to that of a dissemination pipeline in RAPID. Indeed a dissemination pipeline is simply one form of virtual world, with a regular application topology. Both represent a collection of services and resources harnessed by one or more users for a common purpose. The important difference between the two is in terms of how they are managed. In RAPID the state of a dissemination pipeline is encapsulated in a *PipelineManager*: in the vGrid the state of a virtual world is published in the directory service. The RAPID approach was born of the vGrid experience, and the rationale behind this difference is reviewed in section 7.4.2. The most important point to note about a *World* is that it is formed as a collection of *Datasets*.

Another significant difference between the two schema is due to the way clients access these datasets. vGrid has no well-defined notion of a pipeline operator that can be used to encapsulate resources, such as computational processes or data access mechanisms. Instead it simply has datasets: a weakly defined concept with no standard runtime interface<sup>2</sup>. Clients access and share datasets through one or more

---

<sup>2</sup>The weak definition of datasets is a strength, and is what allows the vGrid to integrate so many



<p><b>Site</b></p> <p>organisation : String</p> <p>location : String</p> <p>networks : set&lt;name of Network&gt;</p> <p>manager : name of Person</p> <p>users : set&lt;name of Person&gt;</p> <p>hosts : set&lt;name of Machine&gt;</p> <p><b>Machine</b></p> <p>site : name of Site</p> <p>address : set&lt;INetAddr&gt;</p> <p>architecture : String</p> <p>operating-system : String</p> <p><b>Person</b></p> <p>site : name of Site</p> <p>firstname, surname, username : String</p> <p>email : String</p> <p>credentials</p> <p>icon : URL (<i>download bitmap</i>)</p> <p>avatar : URL (<i>download geometry</i>)</p> <p><b>Network</b></p> <p>name : String</p> <p>organisation : String</p> <p>start-sites : set&lt;name of Site&gt;</p> <p>end-sites : set&lt;name of Site&gt;</p> <p>bandwidth : tuple&lt;min bps, mean bps, max bps&gt;</p> <p>latency : tuple&lt;min msec, mean msec, max msec&gt;</p> <p>packet-loss : tuple&lt;min %, mean %, max %&gt;</p> <p><b>AdaptorFactory</b> machine : name of Machine</p> <p>rmi-reference : Java RMI URL</p> <p>corba-reference : stringfied CORBA reference</p> <p>instances : set&lt;name of Dataset&gt;</p> <p><b>Adaptor</b> name : String</p> <p>type : name of Type</p> <p>source : name of DatasetFactory</p> <p>from : pair&lt;protocol-id, type-id&gt;</p> <p>to : set&lt;pair&lt;protocol-id, type-id&gt;&gt;</p>	<p><b>World</b></p> <p>name : String</p> <p>description : String</p> <p>creator : name of Person</p> <p>icon : URL (<i>download bitmap</i>)</p> <p>model : URL (<i>download geometry</i>)</p> <p>machines : set&lt;name of Machine&gt;</p> <p>members : set&lt;name of Person&gt;</p> <p>dataset : set&lt;name of Dataset&gt;</p> <p><b>Dataset</b></p> <p>name : String</p> <p>type : name of Type</p> <p>source : name of DatasetFactory</p> <p>source-id : String</p> <p>protocols : set&lt;pair&lt;protocol-id, parameters&gt;&gt;</p> <p>adaptors : set&lt;name of Adaptor&gt;</p> <p><b>Type</b></p> <p>description : String</p> <p>type-id : String</p> <p>java-class : URL (<i>download .class file</i>)</p> <p>python-class : URL (<i>download .py file</i>)</p> <p>bamboo-module : URL</p> <p><b>Protocol</b></p> <p>description : String</p> <p>protocol-id : String</p> <p>java-class : URL (<i>download .class file</i>)</p> <p>python-class : URL (<i>download .py file</i>)</p> <p>bamboo-module : URL</p> <p><b>DatasetFactory</b> description : String</p> <p>produces : set&lt;pair&lt;protocol-id, type-id&gt;&gt;</p> <p>machine : name of Machine</p> <p>rmi-reference : Java RMI URL</p> <p>corba-reference : stringfied CORBA reference</p> <p>instances : set&lt;name of Dataset&gt;</p>
--	--

Table 7.1: The vGrid Resource Schema

---

*Protocols.* In addition each Dataset is typed according to a set of known *Types* published in the directory. RAPID offers only a simple enumeration of data types through the implementation profile and assumes use of a common protocol (RAPPORT). vGrid requires that data types and protocols be described in the schema and published in the directory. This affords a degree of introspection and allows applications and traders to dynamically determine the set of all possible data types and sharing protocols. It also allows extensible applications to dynamically download code and components to handle unknown data types or support new protocols. Furthermore, the clean separation of data type from network protocol is a powerful and useful concept.

The virtual world entries in the directory include all the details required to connect a client to a world. Each *World* consists of one or more Datasets, where a dataset might be anything from a multicast set of avatar (object) positions, to streamed voice messages or results of a simulation process. Each Dataset is an instance of a data Type, produced by a Factory, and accessible by one or more Protocols. The dataset is advertised within the directory, along with the connection parameters for every protocol by which it is shared. There may also be Adaptors associated with a dataset to act as a bridge between protocol boundaries. To connect a client to a virtual world it is necessary to retrieve the connection parameters of one protocol for every dataset in the world.

The vGrid supports refinement and transformation of datasets through use of adaptors. An adaptor is an object that converts between one or more data types or network protocols. One important use of adaptors is to support interoperability between client applications: acting as a bridge between data sharing protocols or mapping between equivalent data types. The other important use of adaptors is to refine data types to support visual representations. In this second role an adaptor can be used to convert an abstract dataset into one that can be rendered: a role similar to a visualisation server acting for an HPC simulation. In one sense a dissemination pipeline is simply a collection of adaptors applied to a raw dataset.

Unlike RAPID, the vGrid also defines a standard encoding of its schema within a directory service or namespace. This encoding is demonstrated in Table 7.2. The top level of the namespace is organised into sites, networks, data types, protocols, dataset factories, adaptor factories and virtual worlds. Each site contains sub-directories for people and machines. Each factory has subdirectories to contain all current instances. Other elements of the schema are encoded as attributes of these basic sets of entries. This is a very simple encoding and namespace design, the implications of which are considered in section 7.4.4.

### 7.3.2 Factories and Traders

The vGrid uses a factory mechanism to produce instances of a data type, and perform resource management. Access to datasets follows a period of service negoti-

---

existing applications with such ease.

---

```

/ site      / anu.edu.au      / person      / sam
              brian
              / machine      / upside
              palm
              bondi

/ network   / ebn-syd-canb
              / ebn-canb-melb
              / apan-canb-tokyo

/ type      / nbody-simulation
              / object-position

/ protocol  / cavern-nbody
              cavern-avatars
              http

/ data      / nbody-factory   / instances   / bondi-8954
              bondi-8863
              / position-factory / instances   / palm-2378

/ adaptors  / nbody-adaptor   / instances   / palm-2397

/ worlds    / sams-nbody-world

/ site
  / anu.edu.au,
    organisation= Australian National University
    location=     Acton, Canberra, Australia, 0200
    networks=     ebn-syd-canb, ebn-canb-melb,
                  apan-canb-sing
    manager=      brian

  / person
    / sam,
      firstname=Samuel
      surname=Taylor
      username=sam
      email=sam.taylor@anu.edu.au
      icon=http://acsys.anu.edu.au/~sam/icon.gif
      avatar=http://acsys.anu.edu.au/~sam/sam.vrml
    / brian, ...

```

---

**Table 7.2:** Encoding the vGrid resource schema. The namespace hierarchy is depicted above the line, with token entries for a N-Body simulation world. Below the line are the complete entries for an organisation and a person.

---

ation, and is usually bound by time constraints. However, the role of factories is quite free form within the vGrid, since there is no well-defined service life-cycle, and a more restricted form of service tickets. The negotiation interface to factories is also simpler. Because there is no manager associated with a virtual world, factories assume a greater responsibility for managing datasets and their entries in the directory. Finally, the distinction between dataset factories and adaptor factories is pronounced.

The vGrid supports a much richer variety of *Trader* services than RAPID. It includes basic "world builder" traders – equivalent to the PipelineTrader – to construct new virtual worlds on behalf of a user. It also supports traders to offer lobby services, or visualise portions of the resource schema. A lobby service essentially just allows a client to visualise the set of worlds currently published in the directory. Other schema entries may also be of interest. Examples include an avatar trader to visualise the collection of users (people) recorded in the schema, or a network trader to provide a visual network weather service [163]. Finally traders are used as a vehicle for isolating legacy clients from the underlying grid services. Connection traders are used to interact with the directory on behalf of VE clients, retrieve configuration details and connect a client to a virtual world.

### 7.3.3 Implementation

A prototype implementation and test-bed environment were developed in the Virtual Environments Laboratory at the CRC for Advanced Computational Systems. The test-bed was established in June 1999 and a broad range of applications have been integrated into the grid since then. The success of the prototype speaks strongly for the generality and flexibility of the vGrid.

The prototype implementation was developed as a framework for running existing VE applications. It comprises: a simple directory service used to store the vGrid resource schema; an extensive client-side interface to the directory for use by advanced clients; a highly generic factory for datasets; an adaptor factory; and three trader services. These components form a basic framework into which VE applications and resources are integrated. Their implementation consists of approximately 9000 lines of Java code with an additional 1200 lines of shell scripts. Integrating a new data type into the grid requires minimal if any changes to the factories. Integrating a new client into the grid usually requires nothing more than a simple wrapper script. The prototype is highly portable, and runs on most flavours of Unix as well as Win32 platforms.

A test-bed environment was established to evaluate how well the prototype could support the existing applications and data resources used in the VE Laboratory. The test-bed has access to a variety of high-end local visualisation hardware including Haptic Workbenches [132] and immersive Wedge environments [41, 161]. It also has connections to other national and international laboratories through networks such as the Asia Pacific Advanced Network (APAN) [4]. The test-bed was populated with a range of existing applications developed within the laboratory, and by international

<b>Categories of Virtual World</b>	<b>Data types</b>
Digital terrain visualisation	Object positions
N-body simulation	Digital Terrain Models
Virtual chat room	N-body simulation results
SC'98 demonstration worlds	Text chat messages
Ping-pong game	Audio/speech streams
<b>VE Clients</b>	<b>Dataset Factories</b>
Comet terrain viewer	Digital terrain repository
Avango terrain viewer	Parallel N-body simulation scheduler
Velocity vChat client	N-body logging service
Limbo client	Avatar position sharing manager
AVS module	
VRML browser	<b>Adaptor Factories</b>
Ping-pong game	N-body adaptor
	Digital terrain adaptor
	ObjectPosition adaptor

**Table 7.3:** Elements of the vGrid test-bed

collaborators. Table 7.3 summarises elements of the test-bed. These include the Comet terrain visualisation tool, a collaborative terrain visualisation client, a parallel N-body simulation process, the Supercomputing '98 i-Grid demonstrator [13], a virtual ping-pong game [79] and others. Some support is also offered for a range of general-purpose clients, including VRML browsers and an AVS [136] module. In all cases these applications and visualisation tools have been integrated into the vGrid without requiring any significant changes to code.

One of the more interesting applications, with particular relevance to RAPID, is an N-body simulation world. This allows participants to view and interact with an N-body simulator running on a high-performance parallel computing facility. The simulation shares its results in real time, through the CAVERNsoft protocol. Users view the simulation and can control some parameters through a modified version of the Limbo client. An N-Body factory is used to create an instance of the simulation. A logging service is also available to record the results of each iteration of the simulator, and allow users to replay previously recorded simulations.

Because most applications have not been modified to support the vGrid, the test-bed supports only limited application interoperability. In general, each category of virtual world uses a unique communications protocol and VE client. Three adaptors have been implemented for the test-bed to bridge these protocol boundaries. The N-body simulation is normally viewed with the Limbo client, but an N-body adaptor can be used to share the simulation with VRML browsers. A second adaptor is available to convert Comet terrain models into VRML for use by a range of clients. Work

---

has also started on an object position adaptor to reflect and share the positions of objects within a virtual world across protocol boundaries. Since object movement is the basis of most CVEs [78] the development of this adaptor could be a major step toward greater interoperability.

The test-bed also includes two simple Traders. The World Manager trader provides a means of creating and destroying worlds on behalf of a user. It negotiates with factories to gain access to datasets, and advertises newly created worlds in the directory so that other users may join. The Connection trader connects non-vGrid aware clients to worlds registered in the directory. It completely hides the vGrid infrastructure from the client, but relies on the user knowing the name of a world he or she wishes to join. This is analogous to a web user knowing the URL of the page they want to start browsing from. It also underscores the value of advertising virtual worlds in a common name space. A partial implementation of a Lobby trader has also been made.

## **7.4 Analysis of vGrid**

The vGrid is a generalised form of the application management framework in RAPID. Analysis of the vGrid test-bed provides a means of validating RAPID in use with real applications. This analysis starts by reviewing how well the vGrid meets its stated goals. It then considers issues of virtual world management, configuration, schema encoding and interoperability all of which inform RAPID.

### **7.4.1 Success of vGrid**

The success or failure of a metacomputing environment is difficult to quantify. However, the vGrid project has been very successful in meeting its stated objectives. A large number of disparate applications run within the grid, demonstrating that a common management framework is practical. The vGrid is a useful tool for users wanting to create, advertise, find and join virtual worlds. Supporting this functionality requires very little change to existing applications and does not require a common network protocol or interface. Instead, the grid provides a standard approach to application configuration and resource discovery and management. The vGrid also offers a starting point for problems of interoperability and topology construction, though these remain hard problems in their own right.

### **7.4.2 Virtual World Management**

One weakness of the vGrid is its lack of a manager entity with each virtual world. An underlying assumption of the vGrid is that virtual worlds may be very long-lived entities. Although some virtual worlds use valuable computational and network resources, many do not. Persistent virtual worlds – available 24 hours a day, seven days a week – are increasingly common because the cost of preserving world state is low.

With peer-based CVEs the clients that connect to a world provide the resources required to maintain it. With many client-server based CVEs the amount of state that must persist in the absence of clients is small. Consequently, the lifetime of a virtual world tends to be much longer than that of a dissemination pipeline built from precious computational and network resources.

Given this long world lifetime, world management appears to be an occasional activity rather than a regular event. The underlying philosophy of the vGrid is that the presence or absence of a single dataset should not necessarily affect an entire world. Each factory manages the life-cycle of the resources it produces. So if a factory decides to reclaim a dataset then that should not automatically entail destruction of the world. Consequently, it seemed unnecessary to force worlds to include a manager component; even wasteful for persistent worlds. Instead, external traders are provided for times when management is required.

This approach works well for most of the virtual worlds in the test-bed, but not all. The N-body simulation world uses valuable high-performance computing resources, which cannot be committed on a permanent basis. Although a world can persist after the simulation process ends, it ceases to be meaningful; with users left to navigate and interact in a void space. If simulation results have been logged then there may be value in the users reviewing the log to observe interesting interactions. However, this requires non-trivial management of the world to replace the simulation process with a log playback service.

Another limitation due to lack of a manager is that all world state must be advertised and preserved in the directory. This requirement works better for some data types than others. Advertising the IP address of a multicast group is trivial, but describing a topology of connections is not. It also has significant security implications: even with careful assignment of permissions, the vGrid approach is not designed to be secure. Finally, the world entries in the directory must be accessed and written to by many different actors: clients, factories and traders. A holistic approach is preferable.

Given the experience of the vGrid, the much greater need to manage resources, the short life and the well-defined life-cycle of a dissemination pipeline, RAPID employs a dedicated pipeline manager.

### **7.4.3 Static and dynamic configuration**

The vGrid stores all the infrastructure metadata associated with Collaborative Virtual Environments<sup>3</sup>. This metadata is used to configure and run what are complex and widely distributed applications. So the vGrid can be thought of as an application configuration service.

---

<sup>3</sup>Chapter 3 emphasises a distinction between infrastructure metadata and application metadata. The vGrid stores infrastructure metadata: data about applications and the environment in which they run. Application metadata – descriptive data specific to a single application or domain – is beyond the scope of the vGrid.

---

It is useful to draw a distinction between static and dynamic configuration. Applications with a static structure need be configured only when they begin. A dissemination pipeline is a good example of an application with static structure: the connections between operators are established once, and remain thereafter for the life of the pipeline. Applications with dynamic structure change over time, and so require reconfiguration. The lifetime of an application, and requirements for fault tolerance, often determine whether it has a static structure. Short lived applications can be static: long lived and fault-tolerant applications need to be dynamic to respond to changes in their runtime environment.

The vGrid does an excellent job of configuring static applications, but is less successful with applications requiring reconfiguration. This is due not to a limitation of the grid, but rather to the way applications have been integrated with it. Few applications have been modified to interact directly with the grid. In most cases the Connection trader is used to isolate an application from the grid, retrieve configuration details and present them in an appropriate format. This approach works very well for static configuration but fails for dynamic configuration. Consequently, problems such as the topology of the Supercomputing '98 VE remain, since they require major changes to the basic application. Dynamic configuration of real-time and multimedia applications remains a difficult and open research problem [88].

Dynamic configuration is not required by RAPID. As already noted, dissemination pipelines are static in structure. Their topology and configuration can be established and optimised once, and will remain efficient thereafter. Consequently the vGrid and RAPID approach to application management is highly appropriate.

#### 7.4.4 Schema Encoding

The vGrid schema encoding has scalability limits. In the test-bed environment this was not a problem, but for a larger grid it may be. The scalability problems arise because sites, data types, protocols, factories and worlds are all advertised in a flat name space. As the number of sites, for example, increases so does the cost to a directory server of accessing and maintaining the namespace.

The obvious answer is to arrange the namespaces into hierarchies. A hierarchy of sites could be produced simply from the DNS namespace. Type and protocol entries are less likely to cause scalability problems, and modest hierarchies could be produced from the MIME-style encoding format. Factories and worlds fit less obviously into a hierarchy.

This is hardly an insurmountable problem: but one that needs to be considered in any large-scale deployment of the vGrid. For RAPID it is reasonable to treat encoding as an implementation detail, specified in an implementation profile.

#### 7.4.5 Interoperability

One of the more ambitious goals of the vGrid is to promote interoperability between virtual environments. This is achieved through the use of adaptors to bridge be-



tween protocol boundaries. Only three adaptors have been tried in the test-bed environment: one to export N-body simulation results into a VRML client; one to export Comet terrain models to an Avango-based visualisation client [145]; and one to share the position of moving objects within a virtual space.

Development of these adaptors underscores the difficulty in making disparate virtual environments interoperable. The N-body simulation has to greatly compromise temporal resolution: changes in the simulation being visible less frequently to VRML clients than to Limbo clients. The terrain adaptor has to compromise spatial resolution, since VRML based clients have no ability to make tile-oriented requests. The object position adaptor is able to map positional changes with passable accuracy but only for a small number of applications. It does not scale well either: adding considerable latency as the number of moving objects increases.

Even allowing for the limited number of adaptors in the test-bed, interoperability remains a major problem. Generally speaking clients on one side of the protocol boundary will suffer a diminished experience. This may be acceptable to allow passive observers some sense of what is happening in a virtual space. Interoperability between arbitrary client applications would require a considerable number of different adaptors be developed. However, where a small number of applications are used – as with the terrain visualisations – the number of adaptors can remain manageable.

Differences in protocol are not a major problem: differences in the semantics of data types are. For example, if two applications both use dead-reckoning for object positions, it is no great difficulty to bridge between protocols (say DIS [68] and CAVERNsoft [84]) to support position sharing. A counter example comes in the form of the terrain visualisation adaptor. Here the semantics of the data structures are quite different: one client decomposes a dataset into a tile hierarchy; the other treats the dataset in entirety. In the first example an adaptor must simply match the syntax of data sharing, in the second it must match the semantics.

Though the vGrid doesn't solve the interoperability problem, it does at least offer a systematic approach to promoting cross-application data sharing. RAPID takes a higher level approach to interoperability by defining a standard set of interactions between operators through RAPPORT, and a protocol-independent abstraction for communications.

## **7.5 Summary**

The vGrid performs application management for Collaborative Virtual Environments. It is essentially a superset of the RAPID management framework and demonstrates that the mechanisms developed in section 4.2 are feasible and effective. An experimental vGrid test-bed has been used to successfully integrate and manage a diverse set of virtual environments, and demonstrates how a data-centric grid can be formed.

The vGrid is not without limitations. A scalable schema encoding needs to be

---

developed and dynamic reconfiguration of applications is only possible if client executables are modified extensively. These issues all informed the design of RAPID. Most significant is the issue of virtual world management. The limited world management offered by the vGrid motivates the use of a `PipelineManager` in RAPID.

These minor caveats aside, the vGrid provides a powerful demonstration of how large, distributed applications can be managed and used effectively.



---

# Conclusions and Future Work

---

## 8.1 Summary

This thesis develops a software architecture, known as RAPID, to enable collaborative visualisation of geospatial imagery in Virtual Environments. Earth observation data is very valuable, and is used in many different fields including meteorology, defence, scientific analysis, urban planning and insurance. Collaborative Virtual Environments are rich display devices with the added attraction of human interaction. The value of combining the two is very great. However, before this potential can be realised a range of performance and management problems must be addressed.

The thesis addresses four fundamental requirements: for client *responsiveness*, for high rates of processing and dissemination *throughput*, for *collaborative data sharing* and for *application management*. Responsive data access is, perhaps, the most essential of all requirements. The size and processing requirements of geospatial imagery can impose many delays, yet real-time interaction in a CVE requires a high rate of response. Large size also brings a need for efficient dissemination, from archive to client, at high rates of throughput. User interactions and collaborations add a further requirement to share data with appropriate causal ordering. Finally, since imagery dissemination pipelines are complex distributed systems, there is a need to manage their construction and operation.

Given this basic separation of concerns, a solution has been proposed in the form of a software architecture. The RAPID architecture consists of three main elements:

1. A general-purpose dissemination pipeline is used to access and process imagery. The pipeline uses parallel processing and parallel streaming techniques to achieve high rates of throughput. Interactions between operators in the pipeline are fully asynchronous, so as to decouple clients from the pipeline and aid client responsiveness. Support for approximations within the pipeline further aids responsiveness.
2. Large, parallel caches are deployed close to visualisation clients, to decouple them from the pipeline and improve responsiveness. These caches provide low-latency access to data tiles, and use parallel streaming to move bulk data efficiently. Site caches schedule access to the pipeline and employ speculative request policies to predict future client requests and ensure the pipeline is al-

ways busy. These policies aid both throughput and responsiveness. Finally, caches provide an inter-site group communication service, which is the basis of collaborative data sharing. These ideas are all encapsulated in a single component, the RACE, which is the centrepiece of the RAPID architecture.

3. The final requirement, for application management, is performed in the context of a computational grid. The basic resource discovery problem is solved by publishing resources in the grid directory using a standard schema. Resource management is defined in terms of a standard life-cycle for dissemination pipelines, with discrete phases of service negotiation, initialisation, provision and termination. A standard negotiation interface is defined for resource access, and reservations are made using service tickets. A collection of actors and traders round out the management infrastructure. These include: operator factories to build individual operators and manage a single resource, trader services to construct pipelines, and manager services to provide client access to pipelines.

A range of experimental and production systems have been developed to evaluate different aspects of RAPID. The Comet visualisation system, reviewed in Chapter 5, was used to evaluate responsiveness issues. It demonstrates the viability of parallel caching and downstream gathering, and also highlights the importance of asynchronous communications and speculative requests. Three different imagery dissemination systems (OATS, IMAD-1 and CROP) were reviewed in Chapter 6, which explore ways to overcome throughput constraints. They also underscore the need for application management with widely distributed systems. The vGrid project demonstrated how this could be performed in a computational grid, with extensions for data-centric applications such as Virtual Environments. Collectively, the experimental work motivates and validates the RAPID architecture.

## 8.2 Future Directions for Research

There are always unsolved problems, and RAPID is not the last word in imagery dissemination. Indeed, one of the exciting things about this area of research is the number of interesting problems that remain. One contribution of this thesis is to provide a foundation for future research projects.

### 8.2.1 A Full Implementation of RAPID

Taken as a whole, the five systems in the case studies explore all the major elements of RAPID. However, each system is essentially a subset of the architecture, and as yet a full implementation has not been completed. An implementation in Java is currently being developed, and although progress is promising, it is not mature enough to warrant reporting in this thesis. The value of a full implementation is not in verifying the architecture (since the experimental work has substantially done that), but rather in exploring the boundary between architecture and implementation profile.

---

The importance of separating general concepts from application specific detail was stressed in section 4.1.2. Unfortunately this separation is not always perfect, especially when high levels of performance and optimisation are required. Developing a full implementation of RAPID would, therefore, be a useful and interesting exercise.

### **8.2.2 Evaluating Scheduling and Prefetch Policies**

One of the keys to meeting client responsiveness is to anticipate future client requests. Four different scheduling and prefetch policies were identified in section 3.1.2, to be implemented in RACE caches. Experimental work in Chapter 5 emphasised the need for these techniques, and also demonstrated the value of the Request by Proximity policy. Work with OATS, and systems not reported here [78, 79], demonstrates that the other policies are also valuable. However, a detailed evaluation of prefetching has not yet been performed. Any such evaluation would be highly dependent on a particular implementation of RAPID: the policies that work well for one type of imagery or client, may not work well for others. So this is quite a general problem, and one which would make an interesting thesis in its own right. As such, it is beyond the scope of this thesis, but is a priority for future work by the author.

### **8.2.3 High Level User Interaction and Collaboration**

Of the four fundamental requirements identified in Chapter 1, collaborative data sharing has received perhaps the least attention. This is due, in part, to the fact that it is now a well understood problem [126]. Numerous design trade-offs were identified in Chapter 2, and the solution that was included in RAPID is a minimal data sharing service. This provides a basic foundation for building collaborative systems, but many higher-level usability questions remain. What types of communication and interaction take place between users as they explore a large dataset? What do different types of user look for in a dataset? How can we exploit this to improve both application performance, and user experiences? Once again, the answers to these questions will depend greatly on implementation and on dataset. For example, the interactions between military personnel, with a clear chain of authority, may be quite different to those in an ad-hoc group of scientists. These higher level issues are obviously beyond the scope of this thesis, but are an interesting direction for future research.

### **8.2.4 Other Issues**

Security is important in any distributed system, and is especially significant when sensitive or valuable data is shared. Even though section 1.4 defined security as beyond the scope of this work, it remains a relevant problem.

Another interesting issue is to consider how other types of spatial data can be integrated into the visualisation process. RAPID is specialised to the needs of terrain rendering, based on texture imagery and elevation models (2.2.4). Both imagery and

elevation models are typically represented as uniform arrays of colour or height values. Other forms of spatial data, such as geometric representations of buildings and roads, are represented in quite different ways. Integrating such data structures into a terrain visualisation is an interesting problem, with implications for both the RACE cache and the application management infrastructure.

### **8.3 Conclusions**

RAPID is a general architecture which enables very large geospatial images to be visualised in Collaborative Virtual Environments. It address four fundamental performance and management requirements through a range of streaming, caching and metacomputing techniques. It is a blueprint for a family of responsive imagery dissemination systems, and a foundation for much future work.

---

# **Appendices**

---



---

# RAPID Class Reference

---

## A.1 Service Negotiation Classes

**Class:** Filter  
**Stereotypes:** «remote»  
**Extends:** ServiceInstance ← Operator ← Filter  
**Life-cycle:** initialisation, provision, termination  
**Description:** A Filter is a particular class of Operator in a dissemination pipeline which performs image processing or transformation. It is produced by a FilterFactory when a PipelineManager redeems a ServiceTicket that was promised during an earlier service negotiation.  
**See also:** Section 4.2.1, page 68  
**Interactions:** FilterFactory, PipelineManager

**Class:** FilterFactory  
**Stereotypes:** «remote»  
**Extends:** ServiceProvider ← OperatorFactory ← FilterFactory  
**Life-cycle:** negotiation, initialisation  
**Description:** A FilterFactory is a particular class of OperatorFactory which produces Filter operators for use in a dissemination pipeline. They participate in service negotiation and service initialisation just like any other factory.  
**See also:** Section 4.2.1, page 68  
**Interactions:** Filter, PipelineTrader, PipelineManager, ServiceTicket

**Class:** ImageAccessor  
**Stereotypes:** «remote»  
**Extends:** ServiceInstance ← Operator ← ImageAccessor  
**Life-cycle:** initialisation, provision, termination  
**Description:** An ImageAccessor is a particular class of Operator which retrieves data from an ImageryArchive. As such it is a source of data for a dissemination pipeline. It is produced by the ImageryArchive when a PipelineManager redeems a ServiceTicket that was promised during an earlier service negotiation.

**See also:** Section 4.2.1, page 68  
**Interactions:** ImageryArchive, PipelineManager

**Class:** ImageryArchive  
**Stereotypes:** «remote»  
**Extends:** ServiceProvider ← OperatorFactory ← ImageryArchive  
**Life-cycle:** negotiation, initialisation  
**Description:** An ImageryArchive is a particular class of OperatorFactory which provides an interface to a large repository of geospatial images. It will almost always be a wrapper around an existing interface, such as the GIAS [149]. Archives produce ImageAccessor operators for use in a dissemination pipeline and participate in service negotiation and service initialisation just like any other factory.

**See also:** Section 4.2.1, page 68  
**Interactions:** ImageAccessor, PipelineTrader, PipelineManager, ServiceTicket

**Class:** OperatorFactory  
**Stereotypes:** «remote»  
**Extends:** ServiceProvider ← OperatorFactory  
**Subtypes:** RACEFactory,  
**Life-cycle:** negotiation, initialisation  
**Pattern:** *Abstract Factory* role in the Abstract Factory pattern  
**Description:** An OperatorFactory is used to manage a single resource. They advertise themselves in the resource schema, and are contacted by Traders during the negotiation phase. Traders and factories interact through a Service-Negotiation brokering interface. If they can agree to a set of service characteristics, the factory provides the trader with a ServiceTicket and makes an appropriate resource reservation. During service initialisation the ticket is used by a PipelineManager to require the factory to make good on its promise of a service, and produce an appropriate Operator.

**See also:** Section 4.2.1, page 68  
**Interactions:** PipelineTrader, ServiceNegotiation, ServiceTicket, Operator

**Class:** PipelineManager  
**Stereotypes:** «remote»  
**Extends:** ServiceInstance ← PipelineManager  
**Life-cycle:** negotiation, initialisation, provision, termination  
**Description:** A PipelineManager coordinates the life-cycle of every dissemination pipeline. It is produced by a PipelineTrader following a successful service negotiation. The trader provides the manager with a collection of ServiceTicket objects which can be used to build the pipeline during service initialisation. It also provides the manager with instructions about how the

operators should be connected. The manager waits until the agreed time for service initialisation then constructs the pipeline, joining and initialising operators as described in sections 3.3 and 4.3. During the service provision phase, the manager is used to provide a point of first contact for visualisation clients. Clients contact the manager which forwards them on to their nearest RACE cache. Finally, once service has terminated the manager is responsible for cleaning up the pipeline.

**See also:** Section 4.2.1, page 68

**Interactions:** PipelineTrader, ServiceTicket, OperatorFactory, Operator, Connection

**Class:** PipelineTrader

**Stereotypes:** «remote»

**Extends:** ServiceProvider ← PipelineTrader

**Life-cycle:** negotiation

**Description:** PipelineTrader objects are used to simplify and automate service negotiation for client applications. Each trader produces one or more forms of dissemination pipeline for a client. It uses the resource schema in the grid directory to find required resources. It interacts with OperatorFactory objects, through their ServiceNegotiation interface, to make resource reservations. The results of these negotiations are ServiceTicket objects. Once access to all required resources has been arranged, the trader produces a PipelineManager object to handle the initialisation, provision and termination phase of the pipeline life-cycle.

**See also:** Section 4.2.1, page 68

**Interactions:** OperatorFactory, ServiceTicket, PipelineManager

**Class:** RACEFactory

**Stereotypes:** «remote»

**Extends:** OperatorFactory ← RACEFactory

**Life-cycle:** negotiation, initialisation

**Description:** An RACEFactory is a particular class of OperatorFactory which creates RACE caches at sites where visualisation clients run. They participate in service negotiation and service initialisation just like any other factory.

**See also:** Section 4.2.1, page 68

**Interactions:** RACE, PipelineTrader, PipelineManager, ServiceTicket

**Class:** ServiceInstance

**Stereotypes:** «remote»

**Subtypes:** Operator, PipelineManager, RACE, ImageAccessor, Filter, Transporter

**Life-cycle:** negotiation, initialisation, provision, termination

**Pattern:** *Abstract Product* role in the Abstract Factory pattern

**Description:** ServiceInstance is a highly abstract class, at the root of the application management hierarchy, which represents a single provision of a service. Concrete examples of include Filter operators and PipelineManager objects. Every ServiceInstance is produced by a ServiceProvider. It provides methods to return the distinguished name of the Site and Host on which it runs, as published in the grid directory through the resource schema.

**See also:** Section 4.2.1, page 68

**Interactions:** ServiceProvider

**Class:** ServiceProvider

**Stereotypes:** «remote»

**Subtypes:** OperatorFactory, PipelineTrader, RACEFactory, ImageryArchive, FilterFactory, TransportFactory

**Life-cycle:** negotiation, initialisation, provision, termination

**Pattern:** *Abstract Factory* role in the Abstract Factory pattern

**Description:** ServiceProvider is an highly abstract class, at the root of the application management hierarchy, which represents a general class of service. Providers are factories for instances of services. Concrete examples include PipelineTrader and FilterFactory, which produce PipelineManager and Filter objects respectively. Providers are registered in the grid directory, and include methods to return the distinguished name by which they are known.

**See also:** Section 4.2.1, page 68

**Interactions:** ServiceInstance

**Class:** ServiceNegotiation

**Stereotypes:** «remote»

**Life-cycle:** negotiation

**Description:** ServiceNegotiation is the standard interface which clients and traders use to negotiate with an OperatorFactory so that it will produce an Operator. The result of a successful negotiation is not an Operator, but rather a ServiceTicket which can be redeemed at a later date. ServiceNegotiation could also be used as the interface clients use to contact a PipelineTrader, but this is not mandated: more specialised interfaces may be more appropriate for different types of client and trader.

**See also:** Section 4.2.2, page 70

**Interactions:** OperatorFactory, PipelineTrader, ServiceTicket

**Class:** ServiceTicket

**Stereotypes:** «mobile»

---

**Life-cycle:** negotiation, initialisation  
**Description:** The result of a successful negotiation with an operator factory is a *ServiceTicket*. This is a form of *Promise* [91] made by the factory, to produce an appropriate operator at a later date. Tickets are collected by a *PipelineTrader* during the service negotiation phase, and redeemed by the *PipelineManager* during the initialisation phase.  
**See also:** Section 4.2.2, page 70  
**Interactions:** *OperatorFactory*, *ServiceNegotiation*, *Operator*, *PipelineManager*

**Class:** *Transporter*  
**Stereotypes:** «remote»  
**Extends:** *ServiceInstance* ← *Operator* ← *Transporter*  
**Life-cycle:** initialisation, provision, termination  
**Description:** A *Transporter* is a highly specialised form of *Operator* which provides bulk transfer over an experimental or special purpose network. It is used when access to such networks is not transparent or when there are restrictions on the types of traffic that can be sent over the network. Each *Transporter* is produced by a *TransportFactory* when a *PipelineManager* redeems a *ServiceTicket* that was promised during an earlier service negotiation.  
**See also:** Section 4.2.3, page 73  
**Interactions:** *TransportFactory*, *PipelineManager*

**Class:** *TransportFactory*  
**Stereotypes:** «remote»  
**Extends:** *ServiceProvider* ← *OperatorFactory* ← *TransportFactory*  
**Life-cycle:** negotiation, initialisation  
**Description:** An *TransportFactory* is a particular class of *OperatorFactory* which allows access to experimental and special purpose networks. It may be implemented as a wrapper around any existing resource reservation mechanism used by the network. It produces a *Transporter* operator for use in the dissemination pipeline, and participates in service negotiation and service initialisation just like any other factory.  
**See also:** Section 4.2.3, page 73  
**Interactions:** *Transporter*, *PipelineTrader*, *PipelineManager*, *ServiceTicket*

## A.2 Operator and Pipeline Classes

**Class:** *Acceptor*  
**Life-cycle:** initialisation  
**Pattern:** *Acceptor* role in the *Acceptor and Connector* pattern

**Description:** Acceptor objects are used during service initialisation to create the upstream ends of links to ports downstream. Many communication mechanisms require a passive listener to accept new connections. Acceptor performs this role during the initialisation phase of an operators life, when downstream operators try to create a Link to an OutputPort of the operator. Part of this process involves handling OpenLink messages sent from the downstream operator.

**See also:** Section 4.3.2, page 79

**Interactions:** OutputPort, Connection, Link, OpenLink

**Class:** Approximator

**Stereotypes:** «mobile»

**Life-cycle:** initialisation, provision

**Description:** Approximator functions may be associated with the TileBuffer objects of any input or output port. Buffers can use these functions to return an approximation of a tile if the requested tile is not currently in the buffer. Typical approximations might be to return a low-resolution copy of a tile, or through composition or decomposition of other cached tiles which overlap the requested area. Approximator functions are created by Transformation objects and may be passed downstream as part of a Connection object, hence the requirement that they be «mobile».

**See also:** Section 4.4.3, page 87

**Interactions:** Transformation, TileBuffer, Connection

**Class:** Connection

**Stereotypes:** «mobile»

**Life-cycle:** initialisation

**Description:** Connection objects are used to join two operators during service initialisation. A Connection is retrieved from an OutputPort of the upstream operator and passed into an InputPort of the downstream operator. It encapsulates all the logic required to share data between the two operators. This includes:

1. XML descriptions of all instances shared by output (metadata)
2. XML descriptions of tile and LoD hierarchies
3. Factory objects to create different kinds of links (LinkFactory)
4. Link descriptions (name, factory, creation param)
5. An optional map function (RequestMap)
6. An optional approximator (Approximator)

The metadata and tiling information are used to ensure that the two operators are semantically compatible. Link descriptions and link factories are used to create the Link objects used to send data to the upstream operator. A RequestMap may be included if the upstream operator supports

downstream gathering. An *Approximator* may also be included for use on the *InputPort* of the downstream operator.

Connection objects are exchanged during service initialisation by the *PipelineManager*. They are not passed through the pipeline, since it does not exist during initialisation. Instead they are passed across the same object bus used for service negotiation.

**See also:** Section 4.3.2, page 79

**Interactions:** *OperatorPort*, *PipelineManager*, *Link*, *LinkFactory*, *RequestMap*, *Acceptor*, *Approximator*

**Class:** *DependentSet*

**Life-cycle:** provision

**Description:** The *DependentSet* is maintained by the *RequestHandler* within an operator, to record all requests made to upstream operators. When a request is received from a downstream operator it may not be possible to calculate a response without first making a request of upstream operators. The *OutstandingSet* records requests from downstream: the *DependentSet* records the associated requests which have been sent upstream.

**See also:** Section 4.4.4, page 88

**Interactions:** *RequestHandler*, *OutstandingSet*

**Class:** *InputPort*

**Stereotypes:** «remote»

**Extends:** *OperatorPort* ← *InputPort*

**Subtypes:** *RACEPort*, *SlaveInputPort*, *RACESlavePort*

**Life-cycle:** initialisation, provision, termination

**Description:** Data flows along the pipeline enter an operator through one or more *InputPort* objects. Each port is connected to a single upstream operator, and handles flows for a single type of data. During service initialisation the *PipelineManager* provides the port with a *Connection* object to join to the upstream operator. During service provision the port uses one or more *Link* objects send data to and receive data from the upstream operator. There may be more than one *Link* if the other operator supports downstream gathering. If this is the case then the port uses a *RequestMap* function to choose between links when sending *TileRequest* messages. All operator ports have a *TileBuffer* associated with them, and may also use an *Approximator* function to provide temporary results.

**See also:** Section 4.4, page 81

**Interactions:** *PipelineManager*, *Operator*, *Connection*, *Link*, *RequestMap*, *TileBuffer*, *Approximator*

**Class:** *Operator*

**Stereotypes:** «remote»

**Extends:** ServiceInstance ← Operator

**Subtypes:** RACE, ImageAccessor, Filter, Transporter

**Life-cycle:** initialisation, provision, termination

**Pattern:** *Facade* role in the Facade pattern  
*Abstract Product* role in the Abstract Factory pattern

**Description:** The Operator class provides a standard Facade to the various processing and data moving objects at each stage in the pipeline. An operator is a complex entity, with separate subsystems to receive data (OperatorPort), handle signalling (RequestHandler), process requests (Transformation) and buffer results (TileBuffer). The Operator class is a container for all of these different subsystems, and provides a single interface for use during service initialisation.

**See also:** Section 4.2.1, page 68

**Interactions:** OperatorFactory, PipelineManager, OperatorPort, RequestHandler, Transformation, Acceptor

**Class:** OperatorPort

**Stereotypes:** «remote»

**Subtypes:** InputPort, OutputPort, BroadcastPort, SlaveInputPort, SlaveOutputPort, RACEPort, RACESlavePort

**Life-cycle:** initialisation, provision, termination

**Pattern:** *Subject* role in the Observer pattern  
*Observer* role in the Observer pattern

**Description:** Data flows into and out of an operator through OperatorPort objects. Ports define the service provision interface to the operator: the high-performance interface which compliments the object bus interface used for service negotiation and initialisation. OperatorPort is the base of a hierarchy of different ports, with important differences between the input and output ports of an operator. There are a number of characteristics common to all ports. They all use one or more Link objects to send and receive RAPPORT messages. They also have a TileBuffer where bulk data is stored upon receipt from upstream, and prior to delivery downstream. All ports have a name attribute which is unique within the operator, so the tuple (Operator.name, OperatorPort.name) is unique within the pipeline. Finally, ports are typed with a MIME-style encoding, to ensure that connections between operators are meaningful.

Although the RequestHandler contains most of the logic to deal with RAPPORT messages, ports can respond automatically to some messages. For this reason there is a short chain of request handlers formed from Link through OperatorPort to RequestHandler. This chain is based on the Observer design pattern. Since the port sits in the middle of the chain it both observes the link, and is the subject of observation by the handler. For further details please refer to section 4.4.1.

**See also:** Section 4.4, page 81



**Interactions:** Operator, PipelineManager, TileBuffer, Link

**Class:** OutputPort

**Stereotypes:** «remote»

**Extends:** OperatorPort ← OutputPort

**Subtypes:** RACEPort, SlaveOutputPort, RACESlavePort

**Life-cycle:** initialisation, provision, termination

**Description:** Data flows out of an operator through one or more OutputPort objects, in much the same way as it flows in through an InputPort. During service initialisation the PipelineManager retrieves a Connection object from the OutputPort to extend the pipeline downstream. An Acceptor object listens for attempts to open new Link objects from downstream before handing them to the OutputPort. During service provision the port uses these links to listen for requests (TileRequest) and to return results.

There is one important difference between the input and output ports. Although inputs may only be connected to a single upstream operator, outputs may be connected to several downstream operators. This allows forks in the pipelines, and is essential so that more than one client can connect to a RACE cache, and also so that more than one cache can connect to the rest of the pipeline.

**See also:** Section 4.4, page 81

**Interactions:** PipelineManager, Operator, Connection, Link, Acceptor, TileBuffer, Approximator

**Class:** OutputQueue

**Life-cycle:** provision

**Description:** Each operator maintains an OutputQueue to hold results which have been processed but not yet sent downstream. In a responsive system result queueing should be kept to an absolute minimum, and so the OutputQueue is typically very short. In many forms of parallel operator it may be omitted entirely.

**See also:** Section 4.4.4, page 88

**Interactions:** RequestHandler, Transformation

**Class:** OutstandingSet

**Life-cycle:** provision

**Description:** All requests received by an operator are kept in an OutstandingSet until such time as a final response has been calculated. The OutstandingSet is maintained by the RequestHandler and contains all the attributes of the original TileRequest. It records any dependent requests that were made upstream, with the related entries in the DependentSet. The OutstandingSet is also used to detect duplicate and simultaneous requests, and ensure that a response is sent to all requesting downstream operators.

**See also:** Section 4.4.4, page 88

**Interactions:** RequestHandler

**Class:** Link

**Stereotypes:** «active»

**Subtypes:** BroadcastLink

**Life-cycle:** initialisation, provision

**Pattern:** *Abstract Product* role in the Abstract Factory pattern

*Subject* role in the Observer pattern

**Description:** Link is an abstract class which represents a reliable, duplex communication mechanism. It is used to send and receive RAPPORT messages between operators in the pipeline. Link objects are created during service initialisation, after a Connection object has been sent from an OutputPort of one operator to an InputPort of another. At the input of the downstream operator the Link is created by a LinkFactory passed in the Connection. At the output port of the upstream operator, the Link is created by an Acceptor when an OpenLink message is received.

**See also:** Section 4.4, page 81

**Interactions:** LinkFactory, OperatorPort, TileBuffer

**Class:** LinkFactory

**Stereotypes:** «mobile»

**Life-cycle:** initialisation

**Pattern:** *Abstract Factory* role in the Abstract Factory pattern

**Description:** LinkFactory objects are included as part of the Connection sent to a downstream operator when it is joined in to the pipeline during service initialisation. They are used at the operator to construct Link objects as described by the Connection.

**See also:** Section 4.3.2, page 79

**Interactions:** Connection, Link, OperatorPort

**Class:** RequestHandler

**Stereotypes:** «active»

**Subtypes:** RACEHandler, ProxyHandler

**Life-cycle:** provision

**Description:** A RequestHandler is responsible for handling most of the messages received by an operator, and making dependent requests and for coordinating the processing of requests. The asynchronous and multi-user characteristics of the pipeline make the operation of the RequestHandler quite complex. It uses four major data structures to support its operation: an OutstandingSet which is the master list of unanswered requests; a

DependentSet which lists all requests sent upstream for which a response has not been received; a SupportedQueue which is used to feed work to the Transformation function; and an OutputQueue which feeds results back from the transformation ready for delivery downstream.

**See also:** Section 4.4.4, page 88

**Interactions:** Operator, OperatorPort, TileMessage, FlowControl, OutstandingSet, DependentSet, SupportedQueue, OutputQueue, Transformation

**Class:** RequestMap

**Stereotypes:** «mobile»

**Life-cycle:** initialisation, provision

**Description:** A RequestMap may be attached to an InputPort in order to support downstream gathering. When downstream gathering is used to stream data between two operators, the downstream operator's InputPort will have more than one Link to the upstream operator. To send a TileRequest message upstream the port needs a mechanism to select between these links. A RequestMap function provides the port with a mapping of request to link. Not all mapping functions are static in nature, and so MapUpdate messages may be sent along the pipeline to update map functions as required.

**See also:** Section 4.4.1, page 81

**Interactions:** Connection, Link, InputPort, TileRequest, MapUpdate

**Class:** SupportedQueue

**Life-cycle:** provision

**Description:** The SupportedQueue is one of the four data structures used by a RequestHandler to coordinate the processing of tiles. The SupportedQueue is the work queue for the Transformation function within the operator. Entries in the queue represent partially or fully supported requests: i.e. requests for which all required inputs are available or can be approximated. The queue is arranged in priority order, based on the priorities of the original requests.

**See also:** Section 4.4.4, page 88

**Interactions:** RequestHandler, Transformation

**Class:** TileBuffer

**Life-cycle:** provision

**Description:** A TileBuffer is associated with every port to hold the raw data (tiles) used by the operator. Buffers on input ports hold the responses received from upstream operators, which form the inputs to the local Transformation task. Buffers on output ports hold the results after transformation, for delivery downstream.

**See also:** Section 4.4, page 81

**Interactions:** Link, OperatorPort, Approximator

**Class:** Transformation

**Life-cycle:** provision

**Description:** The Transformation function in an operator performs the actual image processing. It runs as an independent thread, receiving work from the SupportedQueue and storing results in the OutputQueue. The Transformation function has the additional responsibility for producing Approximator objects for use on local output and remote input ports. Finally, it is also used by the RequestHandler to determine what dependent requests should be made to satisfy a request. In other words, the handler asks the transformation “to produce this output what inputs do you require?”

**See also:** Section 4.4.4, page 88

**Interactions:** Approximator, RequestHandler, SupportedQueue, OutputQueue

### A.3 RAPPORT Classes

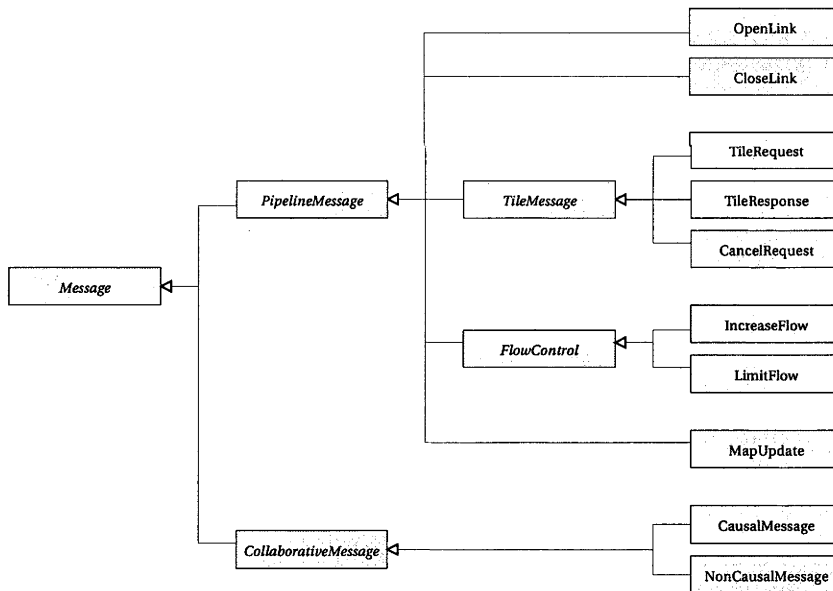


Figure A.1: The complete RAPPORT class hierarchy

**Class:** CancelRequest

**Stereotypes:** «mobile»

**Extends:** Message ← PipelineMessage ← TileMessage ← CancelRequest

**Life-cycle:** provision

**Description:** CancelRequest messages are sent upstream when an operator wants to cancel a tile that was previously requested with a TileRequest message. This is primarily used by RACE caches that make a request to the pipeline in parallel with a request to other caches. If another cache responds before the pipeline, the requesting RACE must cancel the pipeline request using a CancelRequest message.

**See also:** Section 4.4.2, page 84

**Interactions:** RequestHandler, Link, OperatorPort

**Class:** CausalMessage

**Stereotypes:** «mobile»

**Extends:** Message ← CollaborativeMessage ← CausalMessage

**Life-cycle:** provision

**Description:** CausalMessage is one of two classes of collaborative data which is shared over the inter-site bus. It contains data which must be delivered reliably, and in causal order. Such messages are sent through a BroadcastPort to one or more sites or visualisation clients.

**See also:** Section 4.5.5, page 101

**Interactions:** RequestHandler, BroadcastLink, BroadcastPort

**Class:** CloseLink

**Stereotypes:** «mobile»

**Extends:** Message ← PipelineMessage ← CloseLink

**Life-cycle:** provision, termination

**Description:** To support graceful closure of a network link between two operators, either may send a CloseLink message. This results in the termination of the associated Link, and so is typically used during the pipeline termination phase.

**See also:** Section 4.4.2, page 84

**Interactions:** Link, OperatorPort

**Class:** CollaborativeMessage

**Stereotypes:** «mobile»

**Extends:** Message ← CollaborativeMessage

**Subtypes:** CausalMessage, NonCausalMessage

**Life-cycle:** provision

**Description:** There are two separate hierarchies of messages in the RAPPORT protocol: pipeline messages and collaborative messages. The former flow up and down the pipeline between operators. The later are descended from the abstract CollaborativeMessage, and flow across the inter-site formed between RACE caches at each visualisation site. Collaborative messages are all time-stamped to provide a temporal ordering. Subclass of this message may also define causal and non-causal ordering.

**See also:** Section 4.5.5, page 101  
**Interactions:** BroadcastLink, BroadcastPort

**Class:** FlowControl  
**Stereotypes:** «mobile»  
**Extends:** Message ← PipelineMessage ← FlowControl  
**Subtypes:** LimitFlow, IncreaseFlow  
**Life-cycle:** provision  
**Description:** FlowControl messages can be sent along the dissemination pipeline to increase or decrease the rate of requests. Precisely how a RequestHandler adapts its behaviour to these messages is undefined. However, a RACE cache can use flow control messages to moderate the rate at which it makes speculative requests. This class is abstract, and contains only one attribute: a time stamp at which the flow control event occurred.

**See also:** Section 4.4.2, page 84  
**Interactions:** RequestHandler, Link, OperatorPort

**Class:** IncreaseFlow  
**Stereotypes:** «mobile»  
**Extends:** Message ← PipelineMessage ← FlowControl ← IncreaseFlow  
**Life-cycle:** provision  
**Description:** IncreaseFlow messages are sent along the pipeline to indicate that an operator is not running at full capacity. They can be used by the RequestHandler on a RACE cache to increase the rate at which speculative requests are made.

**See also:** Section 4.4.2, page 84  
**Interactions:** RequestHandler, Link, OperatorPort

**Class:** OpenLink  
**Stereotypes:** «mobile»  
**Extends:** Message ← PipelineMessage ← OpenLink  
**Life-cycle:** initialisation  
**Description:** An OpenLink message is sent from a downstream operator when it attempts to open a connection to an upstream operator. A message is sent for each link that must be opened. The contents of the message are largely dependent on the network primitive on which the link is based. OpenLink is the only RAPPORT message used during service initialisation, and is handled by an Acceptor object. Sending this message causes the Acceptor to create an appropriate Link object on the upstream side of the connection.

**See also:** Section 4.3.2, page 79  
**Interactions:** Connection, Link, Acceptor

**Class:** `LimitFlow`  
**Stereotypes:** «mobile»  
**Extends:** `Message` ← `PipelineMessage` ← `FlowControl` ← `LimitFlow`  
**Life-cycle:** provision  
**Description:** `LimitFlow` messages are sent along the pipeline to indicate that an operator is being overwhelmed with requests and so performing below peak efficiency. They can be used by the `RequestHandler` on a RACE cache to decrease the rate at which speculative requests are made. They can also be passed on to visualisation clients to provide users with a warning during times of low responsiveness.  
**See also:** Section 4.4.2, page 84  
**Interactions:** `RequestHandler`, `Link`, `OperatorPort`

**Class:** `MapUpdate`  
**Stereotypes:** «mobile»  
**Extends:** `Message` ← `PipelineMessage` ← `MapUpdate`  
**Life-cycle:** provision  
**Description:** When downstream gathering is used, a `RequestMap` function is required to route request messages upstream. If the mapping function is dynamic, it must be updated from time to time. `MapUpdate` messages are sent down the pipeline to update mapping functions as and when they change.  
**See also:** Section 4.4.2, page 84  
**Interactions:** `RequestMap`, `Link`, `OperatorPort`

**Class:** `Message`  
**Stereotypes:** «mobile»  
**Subtypes:** `PipelineMessage`, `OpenLink`, `CloseLink`, `TileMessage`, `TileRequest`, `TileResponse`, `CancelRequest`, `FlowControl`, `IncreaseFlow`, `LimitFlow`, `MapUpdate`, `CollaborativeMessage`, `CausalMessage`, `NonCausalMessage`  
**Life-cycle:** provision  
**Pattern:** *Serializable* role in the *Serializer* pattern  
**Description:** During service provision, all interactions between operators take place through operator ports using a standard set of messages known as the RAPPOR protocol. `Message` is an abstract class at the root of the RAPPOR signalling hierarchy, with all other messages descended from it. It contains only two attributes: a source and destination address. If no destination is specified then the message is assumed to be for all operators. Messages are «mobile» objects, as described in section 4.1.3. Object mobility can be implemented in various different ways, and if necessary this class can be used to contain packing and unpacking code as described by the *Serializer* [98] design pattern.  
**See also:** Section 4.4.2, page 84

**Interactions:** Link, OperatorPort

**Class:** NonCausalMessage

**Stereotypes:** «mobile»

**Extends:** Message ← CollaborativeMessage ← NonCausalMessage

**Life-cycle:** provision

**Description:** NonCausalMessage is one of two classes of collaborative data shared over the inter-site bus. It contains data which can be delivered unreliably, and with relaxed ordering. Such messages are sent through a BroadcastPort to one or more sites or visualisation clients.

**See also:** Section 4.5.5, page 101

**Interactions:** RequestHandler, BroadcastLink, BroadcastPort

**Class:** PipelineMessage

**Stereotypes:** «mobile»

**Extends:** Message ← PipelineMessage

**Subtypes:** OpenLink, CloseLink, TileMessage, TileRequest, TileResponse, CancelRequest, FlowControl, IncreaseFlow, LimitFlow, MapUpdate

**Life-cycle:** provision

**Description:** There are two separate hierarchies of messages in the RAPPORT protocol: pipeline messages and collaborative messages. The former flow up and down the pipeline between operators during the provision phase of operation. PipelineMessage is an abstract class that contains only two attributes beyond those of the base Message: a destination port and a priority value. Prioritisation is required so that demand driven requests can preempt speculative requests in the pipeline.

**See also:** Section 4.4.2, page 84

**Interactions:** Link, OperatorPort

**Class:** TileMessage

**Stereotypes:** «mobile»

**Extends:** Message ← PipelineMessage ← TileMessage

**Subtypes:** TileRequest, TileResponse, CancelRequest

**Life-cycle:** provision

**Description:** TileMessage is an abstract class from which three specific tile related messages are derived. Request and response messages, the overwhelming majority of traffic in the pipeline, are both descended from this class. The only attribute of TileMessage is a requestID, which is guaranteed to be unique at the requesting (downstream) operator. The tuple (sourceOperator, requestID) uniquely identify every request in the pipeline.

**See also:** Section 4.4.2, page 84



**Interactions:** RequestHandler, Link, OperatorPort

**Class:** TileRequest

**Stereotypes:** «mobile»

**Extends:** Message ← PipelineMessage ← TileMessage ← TileRequest

**Life-cycle:** provision

**Description:** TileRequest messages flow up the pipeline, originating at visualisation clients and RACE caches. Each request can only travel between two operators, but to satisfy a request an operator may have to make dependent requests of its own. So a chain of requests cascade up the pipeline toward the ultimate sources of data: ImageAccessor operators. Each request describes the spatial coordinates of a single tile, the size and level of detail required, the dataset and the visual channel (for multi-channel sources). Requests also specify whether or not approximations should be returned. To support upstream scattering, a request can also specify the return link along which response messages should be sent.

**See also:** Section 4.4.2, page 84

**Interactions:** RequestHandler, Link, RequestMap, OperatorPort

**Class:** TileResponse

**Stereotypes:** «mobile»

**Extends:** Message ← PipelineMessage ← TileMessage ← TileResponse

**Life-cycle:** provision

**Description:** TileResponse messages contain the bulk data that flows down the pipeline from imagery archives to visualisation clients. Each TileResponse corresponds to a previous TileRequest. The payload of each response message is a large tile of imagery. Link objects typically attempt to optimise the receipt of these messages, and store the payload directly into a TileBuffer to avoid copying overheads. Aside from the payload, response messages also include a flag to indicate whether the response is an approximation or a final result.

**See also:** Section 4.4.2, page 84

**Interactions:** RequestHandler, Link, TileBuffer, OperatorPort

## A.4 RACE Classes

**Class:** BroadcastPort

**Stereotypes:** «remote»

**Extends:** OperatorPort ← BroadcastPort

**Life-cycle:** initialisation, provision, termination

**Description:** Each RACE contains a `BroadcastPort` through which it is connected to every other RACE in the pipeline. These ports are the basis of the inter-site sharing bus used for cache snooping and collaborative data sharing. `BroadcastPort` objects only exist on the master RACE node. It is a specialised form of `OperatorPort` which includes only a small `TileBuffer` to hold tiles snooped from other caches, when in transit on their way to a Slave node.

**See also:** Section 4.5.5, page 101

**Interactions:** `BroadcastLink`, RACE

**Class:** `BroadcastLink`

**Stereotypes:** «active»

**Extends:** `Link` ← `BroadcastLink`

**Life-cycle:** initialisation, provision

**Description:** `BroadcastLink` objects are used to send data between `BroadcastPort` objects on the RACE caches at each site. They actually implement the inter-site bus, and so are different to the point-to-point links between ports in the pipeline. They support reliable and unreliable delivery of `CollaborativeMessage` objects between sites. They may be implemented with regular unicast mechanisms (such as a UDP and TCP socket pair), or with a group communication mechanism such as IP multicast.

**See also:** Section 4.5.5, page 101

**Interactions:** `BroadcastPort`, `CollaborativeMessage`

**Class:** RACE

**Stereotypes:** «remote»

**Extends:** `Operator` ← RACE

**Life-cycle:** initialisation, provision, termination

**Pattern:** *Proxy* role in the Proxy pattern

*Master* role in the Master-Slave pattern

**Description:** A RACE is the master node in a parallel RACE cache. It is a subtype of `Operator` and behaves in most respects like any other parallel operator.

**See also:** Section 4.5, page 95

**Interactions:** `RACEFactory`, `PipelineManager`, `RACEPort`, `Slave RACEHandler`, `Scheduler`, `Acceptor`, `BroadcastPort`, `BroadcastLink`

**Class:** `RACEHandler`

**Stereotypes:** «active»

**Extends:** `RequestHandler` ← `RACEHandler`

**Life-cycle:** provision

**Description:** RACEHandler is a simplified form of RequestHandler which runs on the master node of a RACE cache. Most requests are handled automatically by the ProxyHandler objects running Slave nodes. However, slaves cannot make requests up the pipeline. Instead the RACEHandler makes all upstream requests. This includes both demand driven requests from visualisation clients, and also speculative requests generated by the cache Scheduler.

Because no transformation runs on the cache it does not need the same elaborate request sets as other RequestHandler objects. It maintains an OutstandingSet but does not require the DependentSet (since there is a one-to-one mapping between outstanding and dependent requests) nor the SupportedQueue or OutputQueue (since no Transformation runs).

**See also:** Section 4.5, page 95

**Interactions:** RACEPort, Slave, Link, Scheduler, Scheduler, Acceptor, BroadcastPort, BroadcastLink

**Class:** RACEPort

**Stereotypes:** «remote»

**Extends:** OperatorPort ← InputPort ← RACEPort

OperatorPort ← OutputPort ← RACEPort

**Life-cycle:** initialisation, provision, termination

**Description:** RACEPort is a specialised form of OperatorPort which runs on the master node in a RACE operator. It is used during service initialisation to connect the RACE to the end of the pipeline using upstream scattering. It is used by clients during service provision to connect to the Slave nodes using downstream gathering. However, no data passes through a RACEPort. It has a link upstream to make requests, but does not have links downstream. Downstream clients open links to the RACESlavePort on every Slave but not to the RACEPort. The slave ports relay any request they cannot satisfy through an internal communications link. This simplified role for the RACEPort means that it has no TileBuffer or Approximator as other ports do.

**See also:** Section 4.5.1, page 95

**Interactions:** RACE, PipelineManager, RACEHandler, Connection, Link, Acceptor, TileBuffer, Approximator, RequestMap

**Class:** RACESlavePort

**Extends:** OperatorPort ← InputPort ← SlaveInputPort ← RACESlavePort

OperatorPort ← OutputPort ← SlaveOutputPort ← RACESlavePort

**Life-cycle:** initialisation, provision

**Description:** RACESlavePort is a specialised form of OperatorPort which runs on Slave nodes in a RACE operator. It acts as both an InputPort and an OutputPort, with a single TileBuffer used to hold tiles as they flow through the operator. This buffer may have an Approximator associated with it, so that the Slave can return approximations where appropriate.

- See also:** Section 4.5.1, page 95
- Interactions:** `Slave`, `ProxyHandler`, `Connection`, `Link`, `Acceptor`, `TileBuffer`, `Approximator`, `RequestMap`
- Class:** `Scheduler`
- Life-cycle:** provision
- Description:** A `Scheduler` is an implementation of one or more pipeline scheduling policies which make speculative requests for tiles. These are design to both improve responsiveness by anticipating future requests, and to use spare capacity in the pipeline and so increase throughput.
- See also:** Section 4.5.4, page 100  
Section 3.1.2, page 38
- Interactions:** `RACEHandler`, `FlowControl`, `IncreaseFlow`, `LimitFlow`
- Class:** `Slave`
- Life-cycle:** initialisation, provision
- Pattern:** *Slave* role in the Master-Slave pattern
- Description:** A `Slave` node is one single host in a large parallel RACE cache. It communicates to other elements of the pipeline through one or more `RACESlavePort` objects. Only minimal request handling takes place on a slave node, using a `ProxyHandler`. Most serious logic is concentrated in the `RACEHandler` on the Master node.
- See also:** Section 4.5.1, page 95
- Interactions:** `RACEFactory`, `RACE`, `RACESlavePort`, `ProxyHandler`, `Acceptor`
- Class:** `SlaveInputPort`
- Extends:** `OperatorPort` ← `InputPort` ← `SlaveInputPort`
- Subtypes:** `RACESlavePort`
- Life-cycle:** initialisation, provision
- Description:** Parallel operators may need to replicate some parts of their `InputPort` onto multiple `Slave` nodes. A `SlaveInputPort` is a replica for use on a `Slave` node. It may contain a `TileBuffer`, `Approximator`, `RequestMap` and `Link` objects to send data upstream. The precise structure of a `SlaveInputPort` will depend on the model of parallelism adopted within an operator. The `RACESlavePort` is the only concrete form of this port described here.
- See also:** Section 4.4.5, page 91
- Class:** `SlaveHandler`
- Stereotypes:** «active»
- Extends:** `RequestHandler` ← `SlaveHandler`

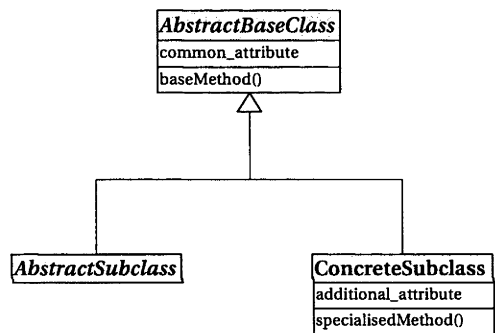
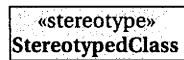
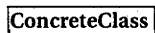
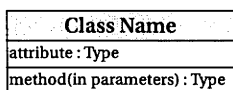
- 
- Life-cycle:** provision
- Description:** ProxyHandler is a simplified form of RequestHandler which runs on Slave nodes. The proxy does not have the ability to make requests upstream. Any TileRequest that cannot be satisfied from the slaves TileBuffer is forwarded on to the RACEHandler running on the master node. The proxy maintains an OutstandingSet but does not require the DependentSet or other queues used by a normal handler.
- See also:** Section 4.5.3, page 99
- Interactions:** Slave, RACESlavePort, Link, PipelineMessage
- 
- Class:** SlaveOutputPort
- Extends:** OperatorPort ← OutputPort ← SlaveOutputPort
- Subtypes:** RACESlavePort
- Life-cycle:** initialisation, provision
- Description:** Parallel operators may also need to replicate some parts of their OutputPort onto multiple Slave nodes. A SlaveOutputPort is a replica for use on a Slave node. It may contain a TileBuffer, Approximator and Link objects to send results back downstream.
- See also:** Section 4.4.5, page 91



# UML Notation and Conventions

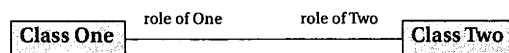
## B.1 Class Diagrams

### Declarations

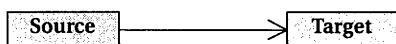


### Associations

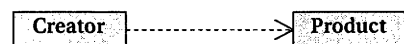
A general association between two classes



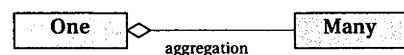
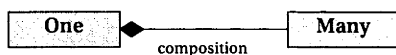
Navigability



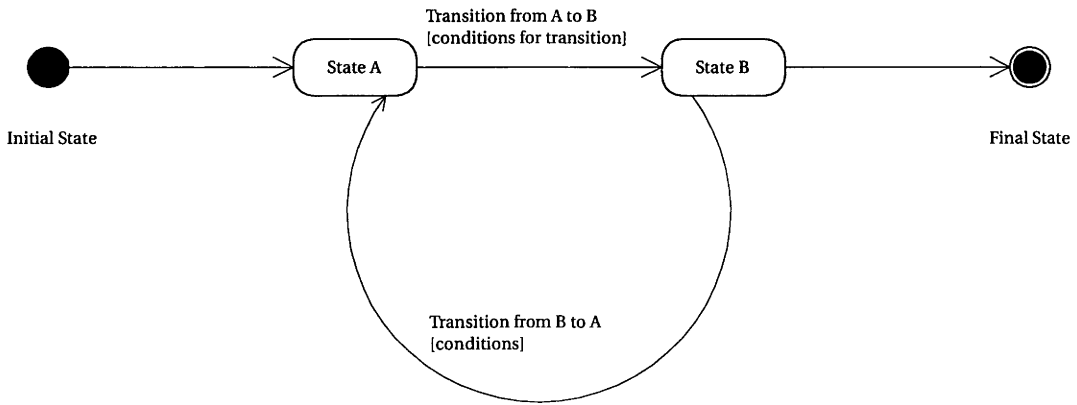
Creation



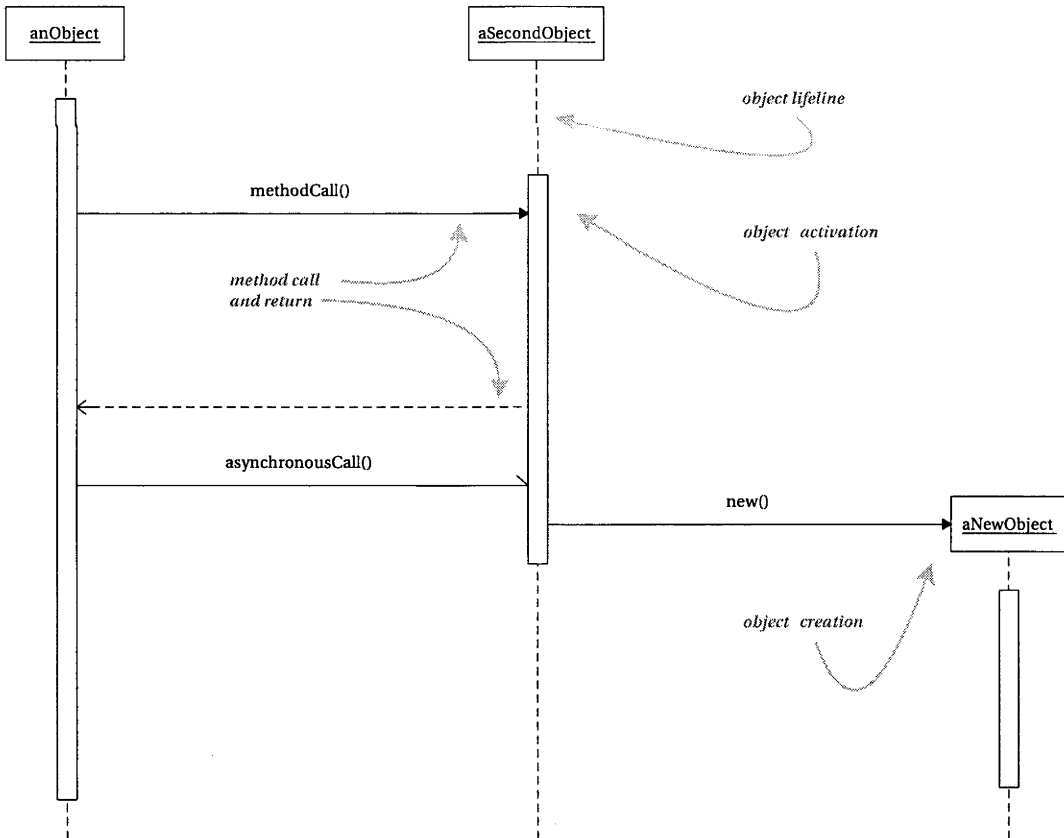
Multiplicities are implied by aggregation and composition



### B.2 State Diagrams



### B.3 Sequence Diagrams





---

# Bibliography

---

- [1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996. (pp. 20, 24)
- [2] Agrecon Pty Ltd. <http://www.agrecon.canberra.edu.au>. (p. 140)
- [3] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996. (pp. 20, 92)
- [4] APAN. The Asia Pacific Advanced Network. <http://www.apan.net>. (p. 153)
- [5] S. Benford and L. Fahlen. A spatial model of interaction in large-scale virtual environments. *3rd European Conference on CSCW*, pages 109–124, 1993. (p. 22)
- [6] Wes Bethel, Brian Tierney, Jason Lee, Dan Gunter, and Stephen Lau. Using high-speed WANs and network data caches to enable remote and distributed visualization. In *SC2000: High Performance Networking and Computing*. Dallas Convention Center, Dallas, TX, USA, November 4–10, 2000, pages 59–59. ACM Press and IEEE Computer Society Press, 2000. (pp. 30, 41)
- [7] Paul Bettner and Mark Terrano. 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. In *Proceedings of Game Developer Conference (GDC) 2001*, March 2001. (p. 21)
- [8] K. P. Birman, R. Cooper, T. A. Joseph, K. P. Kane, F. Schmuck, and M. Wood. *ISIS—a distributed programming environment*. Cornell University, Ithaca, NY, June 1990. in User’s Guide and Reference Manual. (p. 24)
- [9] Kenneth Birman. ISIS: A system for fault-tolerance in distributed systems. Technical Report TR 86-744, Cornell University Computer Science Department, Ithaca, NY, April 1986. (p. 24)
- [10] Stephen Michael Blackburn. *Persistent Store Interface: A foundation for scalable persistent system design*. PhD thesis, Department of Computer Science, Australian National University, August 1998. (p. 28)
- [11] Jonathan Blow. Implementing a Texture Caching System. *Game Developer*, pages 46–56, April 1998. (p. 27)

- [12] Jonathan Blow. Terrain Rendering at High Levels of Detail. In *Proceedings of the Game Developers Conference 2000*, March 2000. (p. 27)
- [13] M. D. Brown, T. A. DeFanti, and M. A. McRobbie et al. The International Grid (iGrid): Empowering Global Research Community Networking Using High Performance International Internet Services. In *Proceedings of INET'99*, San Jose, 1999. (pp. 21, 146, 154)
- [14] Canarie. Canarie. <http://www.canarie.ca>. (p. 73)
- [15] Christer Carlsson and Olof Hagsand. DIVE — A platform for multi-user virtual environments. *Computers and Graphics*, 17(6):663–669, November–December 1993. (p. 19)
- [16] John B. Carter. Design of the munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29(2):219–227, September 1995. (pp. 20, 92)
- [17] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A Hierarchical Internet Object Cache. In USENIX Association, editor, *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA*, USENIX Conference Proceedings 1996, pages 153–163 (or 153–164??), Berkeley, CA, USA, January 1996. USENIX. (p. 28)
- [18] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 2000. (pp. 14, 50)
- [19] D. Chess, C. Harrison, and A. Kershenbaum. Mobile Agents: Are They a Good Idea? *Lecture Notes in Computer Science*, 1222, 1997. (p. 66)
- [20] P. D. Coddington, K. A. Hawick, K. E. Kerry, J. A. Mathew, A. J. Silis, D. L. Webb, P. J. Whitbread, C. G. Irving, M. W. Grigg, R. Jana, and K. Tang. A GIAS-Compliant Server for Geospatial Image Archives. Technical Report DHPC-053, University of Adelaide, Department of Computer Science, University of Adelaide, South Australia, Australia, September 1998. (pp. 13, 138)
- [21] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Programming Design*, volume 1. Addison Wesley, 1995. (pp. 37, 44, 83, 91, 92)
- [22] John De Margheriti and Steve Wang. Micro Forte's Sci Fi Persistent World. In *Proceedings of the Australian Game Developers Conference*, Sydney, Australia, November 1999. (pp. 4, 22)
- [23] Stephen E. Deering. Multicast Routing in Internetworks and Extended LANs. *Computer Communication Review*, 18(4):55–64, August 1988. (p. 101)

- 
- [24] Tom DeFanti and Rick Stevens. *The GRID: Blueprint for a New Computing Infrastructure*, chapter Teleimmersion, pages 131–150. Morgan Kaufmann, 1999. (pp. 18, 21, 30)
- [25] Defence Modelling and Simulation Office (DMSO). *HLA Run-Time Infrastructure: RTI 1.3-Next Generation Programmers Guide*. Department of Defence. (pp. 4, 19, 23)
- [26] Mark A. Duchaineau, Murray Wolinsky, David E. Sigiety, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *IEEE Visualization '97*, October 1997. (p. 26)
- [27] Richard K. Burkard et al. *Geodesy for the Layman*. U.S. National Imagery and Mapping Agency (NIMA), fourth edition edition, March 1984. (p. 17)
- [28] Paul Ferguson and Geoff Huston. *Quality of Service: Delivery QoS on the Internet and in Corporate Networks*. John Wiley and Sons, 1998. (pp. 27, 29)
- [29] Scott S. Fisher and Glen Fraser. Real-time Interactive Graphics in Computer Gaming. *ACM SIGGRAPH – Computer Graphics*, 32(2):15–19, May 1998. (p. 4)
- [30] Peter Fletcher. *Regular Mapping of Multi-Dimensional Data on Parallel Processors*. PhD thesis, Australian National University, Canberra, Australia, May 1993. (pp. 17, 45)
- [31] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, December 1997. (p. 101)
- [32] Australian Centre for Remote Sensing (ACRES). ACRES Digital Catalogue. <http://acs.auslig.gov.au>. (p. 13)
- [33] National Centre for Supercomputing Applications. *Getting Started with HDF – User Manual*. University of Illinois at Urbana-Champaign, 1993. (p. 126)
- [34] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997. (p. 30)
- [35] Ian Foster and Carl Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999. (pp. 8, 30, 48, 54, 67)
- [36] Ian Foster and Carl Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*, chapter The Globus Toolkit, pages 259–278. Morgan Kaufmann, 1999. (p. 30)
- [37] Ian Foster, Carl Kesselman, and Steve Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996. (p. 76)

- [38] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organisations. *International Journal of Supercomputer Applications*, 2001. (p. 30)
- [39] Martin Fowler and Kendall Scott. *UML Distilled Second Edition – A Brief Guide to the Standard Object Modelling Language*. Object Technology Series. Addison-Wesley, 2000. (pp. 9, 65)
- [40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. (pp. 9, 49, 77, 79, 83, 94, 149)
- [41] H.J. Gardner and R.W. Boswell. Effective Virtual Environments - Experiences with a Low Cost Immersive System. In *Proceedings of HPC Asia'98 Conference*, pages 658 – 663, Singapore, 22–25 September 1998. Institute of High Performance Computing. (p. 153)
- [42] Aniruddha S. Gokhale and Douglas C. Schmidt. Measuring the Performance of Communication Middleware on High-Speed Networks. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 26,4 of *ACM SIGCOMM Computer Communication Review*, pages 306–317, New York, August 26–30 1996. ACM Press. (pp. 27, 123, 131)
- [43] Aniruddha S. Gokhale and Douglas C. Schmidt. Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks. In *17th International Conference on Distributed Computing Systems (17th IDCS'97)*, pages 401–410, Baltimore, MD, May 1997. IEEE. (pp. 27, 123, 131)
- [44] Aniruddha S. Gokhale and Douglas C. Schmidt. Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems. In *Proceedings of INFOCOM '99*, New York, March 1999. (p. 36)
- [45] Dean Graetz, Rohan Fisher, Murray Wilson, and Susan Campbell. *Looking Back: The changing face of the Australian continent, 1972 - 1992*. Number 042 in COSSA Publications. CSIRO, 1998. (pp. 1, 12, 14)
- [46] Chris Greenhalgh. *Large Scale Collaborative Virtual Environments*. PhD thesis, Communication Research Group, Department of Computer Science, University of Nottingham, 1999. (p. 22)
- [47] Chris M. Greenhalgh. Awareness Management in the MASSIVE Systems. *Distributed Systems Engineering Journal*, 5(3):129–137, September 1998. (p. 22)
- [48] Mark W. Grigg, A. K. Lui, S. P. James, M. J. Owen, and Edward H. S. Lo. Distributed Imagery Library System Using Java and CORBA. In *Proceedings of Evolve2000*, Sydney, Australia, 2000. DSTC. (pp. 128, 138)

- 
- [49] Andrew S. Grimshaw and William A. Wulf. Legion – A View from 50,000 Feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996. (p. 30)
- [50] Andrew S. Grimshaw, William A. Wulf, and the Legion team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997. (p. 30)
- [51] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1995. (p. 44)
- [52] Object Management Group. *Audio/Video Stream Specification*. Object Management Group, 2000. (p. 27)
- [53] M. Handley and V. Jacobson. SDP: Session Description Protocol, April 1998. Internet Request for Comments 2327. (p. 29)
- [54] K. A. Hawick, H. A. James, C. J. Patten, and F. A. Vaughan. DISCWorld: A Distributed High Performance Computing Environment. In *Proc. of High Performance Computing and Networks (HPCN) Europe '98, Amsterdam*, April 1998. Also available as DHPC Technical Report DHPC-020. (p. 30)
- [55] K. A. Hawick, H. A. James, A. J. Silis, D. A. Grove, K. E. Kerry, J. A. Mathew, P. D. Coddington, C. J. Patten, J. F. Hercus, and F. A. Vaughan. DISCWorld: An Environment for Service-Based Metacomputing. Technical Report DHPC-042, Distributed High Performance Computing Group, Adelaide University, April 1998. (p. 30)
- [56] Ken A. Hawick and Paul D. Coddington. Interfacing to distributed active data archives. *Future Generation Computer Systems*, 16(73), 1999. (pp. 13, 14, 138)
- [57] Ken A. Hawick, Heath A. James, Kevin J. Maciunas, Francis A. Vaughan, Andrew L. Wendelborn, Markus Buchhorn, Michael Rezny, Samuel R. A. Taylor, and Matthew D. Wilson. An ATM-based Distributed High Performance Computing system. In *Proceedings of High Performance Computing and Networks (HPCN) Europe '97, Vienna*, April 1997. (p. 106)
- [58] Ken A. Hawick, Heath A. James, Kevin J. Maciunas, Francis A. Vaughan, Andrew L. Wendelborn, Markus Buchhorn, Michael Rezny, Samuel R. A. Taylor, and Matthew D. Wilson. Geostationary-satellite imagery applications on Distributed, High-Performance Computing. In *Proceedings of High Performance Computing (HPC) Asia '97, Seoul, Korea*, April 1997. (pp. 106, 123)
- [59] High Performance Fortran Forum. *HPF Language Specification, Version 2.0*. January 1997. (pp. 45, 98)
- [60] T. A. Howes, M. C. Smith, and G. S. Good. *Understanding and Deploying LDAP Directory Services*. MacMillan Technical Publishing, 1999. (pp. 29, 73)

- [61] Stuart Hungerford, Dione Smith, and Matthew Wilson. CROP Project Report. Technical Report DHPC-051, University of Adelaide, Department of Computer Science, University of Adelaide, South Australia, Australia, September 1998. (pp. 139, 140, 142)
- [62] Frank E. Redmond III. *DCOM: Microsoft Distributed Component Object Model*. IDG Books, 1997. (p. 123)
- [63] U.S. National Imagery and Mapping Association (NIMA). *Department of Defense World Geodetic System 1984, Its Definition and Relationships With Local Geodetic Systems*. Third edition edition, July 1997. (p. 18)
- [64] U.S. National Imagery and Mapping Association (NIMA). *U.S. Imagery and Geospatial Information System (USIGS) Architecture Framework*. June 1998. (p. 13)
- [65] U.S. National Imagery and Mapping Association (NIMA). *U.S. Imagery and Geospatial Information System (USIGS) Technical Architecture Documentation*. January 1999. (p. 13)
- [66] Space Imaging. <http://www.spaceimaging.com>. (p. 12)
- [67] Institute for Simulation and Training. Dead Reckoning Definitions and Algorithms. In *Enumeration and Bit Encoded Values for Use with Protocols for Distributed Interactive Simulation Applications*. University of Central Florida, 1993. (p. 22)
- [68] Institute of Electrical and Electronics Engineers. *IEEE Standard for Distributed Interactive Simulation-Application Protocols (IEEE-1278)*. IEEE Press, 1995. (pp. 4, 19, 20, 23, 158)
- [69] Interactive Digital Software Association (IDSA). State of the Industry: Report 2000–2001, 2001. <http://www.idsa.com>. (p. 4)
- [70] Internet2. Abilene advanced backbone. <http://www.internet2.edu>. (p. 73)
- [71] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Drive Approach*. Addison-Wesley, 1992. (p. 9)
- [72] Heath A. James. *Scheduling in Metacomputing Systems*. PhD thesis, Department of Computer Science, University of Adelaide, 2000. (pp. 14, 29)
- [73] Heath A. James and Ken A. Hawick. A Web-based Interface for On-Demand Processing of Satellite Imagery Archives. In *Proceedings of the 21st Australasian Computer Science Conference (ACSC'98)*. Springer-Verlag, February 1998. (p. 124)

- 
- [74] Andrew E. Johnson, T. Moher, Jason Leigh, and Y. Lin. QuickWorlds: Teacher driven VR worlds in an Elementary School Curriculum. In *SIGGRAPH 2000 Educators Program*, New Orleans LA, July 2000. (p. 4)
- [75] C. Kanarick. A Technical Overview and History of the SIMNET Project. In *[?] Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, pages 104–111, 1991. (p. 22)
- [76] David Keightley, Ken Tsui, John M. Lilleyman, and Kevin Moore. A Software Framework for an Applications-driven Parallel Image Processing and Display System (PIPADS). In *Proceedings of SPIE Conference on Recent Advances in Sensors, Radiometric Calibration and Processing of Remotely Sensed Data*, Orlando, Florida, April 1993. (p. 14)
- [77] Glenn E. Krasner and Stephen T. Pope. A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988. (p. 19)
- [78] Alex Krumm-Heller. Determinism for Object Position Estimation in Collaborative Virtual Environments. Honours thesis, Department of Computer Science, Australian National University, 1999. (pp. 23, 155, 163)
- [79] Alex Krumm-Heller and Samuel R. A. Taylor. Using determinism to improve the accuracy of dead-reckoning algorithms. In *Proceedings of SimTecT 2000*, pages 243 – 248, Darling Harbour, Sydney, Australia, 2000. (pp. 23, 154, 163)
- [80] Rodger Lea, Kouichi Matsuda, and Ken Miyashita. *Java for 3D and VRML Worlds*. New Riders Publishing, Carmel, IN, USA, December 1996. (p. 22)
- [81] Rodger Lea, Kouichi Matsuda, and J. Rekimoto. Technical Issues in the Design of a Scalable Shared Virtual World. Technical Report SCSL-TR-95-039, Sony Computer Science Laboratories, 1995. (p. 22)
- [82] Yvan G. Leclerc, Martin Reddy, Lee Iverson, and Nat Bletter. TerraVision II: An Overview, 2000. (p. 27)
- [83] V. D. Lehner and T. DeFanti. Distributed Virtual Reality: Supporting Remote Collaboration in Vehicle Design. *IEEE Computer Graphics and Applications*, 1997. (p. 4)
- [84] Jason Leigh. *CAVERN and a Unified Approach to Support Realtime Networking and Persistence in Teleimmersion*. PhD thesis, Electronic Visualization Laboratory, University of Illinois at Chicago, 1998. (pp. 19, 21, 42, 158)
- [85] Jason Leigh, Andrew E. Johnson, and T. A. DeFanti. CAVERN: A Distributed Architecture for Supporting Scalable Persistence and Interoperability in Collaborative Virtual Environments. *Virtual Reality: Research, Development and Applications*, 2(2):217–237, December 1997. (pp. 20, 21, 42, 101, 146)

- [86] Jason Leigh, Andrew E. Johnson, Thomas A. DeFanti, Stuart Bailey, and Robert Grossman. A Methodology for Supporting Collaborative Exploratory Analysis of Massive Data Sets in Tele-Immersive Environments. In *Proceedings of the 8th IEEE International Symposium on High Performance and Distributed Computing (HPDC'8)*, pages 62–69, Redundo Beach, California, August 1999. (p. 20)
- [87] M. Lesk. *Practical Digital Libraries*. Morgan Kaufmann, 1999. (p. 13)
- [88] B. Li and K. Nahrstedt. Dynamic Reconfiguration for Complex Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, June 1999. (p. 157)
- [89] John C. Lin and Sanjoy Paul. RMTP: A reliable multicast transport protocol. *Proceedings of IEEE Infocom*, pages 1414–1424, March 1996. (p. 101)
- [90] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner. Real-time continuous level of detail rendering of height fields. *Proceedings of SIGGRAPH'96*, pages 109–118, 1996. (p. 26)
- [91] Barbara Liskov and S. Liuba. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*. ACM Press, 1988. (pp. 52, 70, 171)
- [92] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - a Hunter of Idle Workstations. In *Proceedings of the IEEE 8th International Conference on Distributed Computing Systems*, pages 104–111, 1988. (p. 29)
- [93] Andrew K. Lui, Mark W. Grigg, T. Andrew Au, and Michael J. Owen. Component Based Application Framework for Systems Utilising the Streaming Data Passing Semantic. In *Proceedings of TOOLS Pacific 2000*, 2000. (pp. 14, 139)
- [94] H. Detmold M. Hollfelder and M.J. Oudshoorn. A Structured Communication Mechanism for Mobile Java Objects as Ambassadors. *Australian Computer Science Communications*, 21(1):265 – 276, February 1999. (p. 66)
- [95] M. R. Macedonia, D. P. Brutzman, M. J. Zyda, D. R. Pratt, P. T. Barham, J. Falby, and J. Locke. NPSNET: A multi-player 3D virtual environment over the Internet. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*. ACM SIGGRAPH, April 1995. (p. 22)
- [96] Michael Macedonia. *A Network Software Architecture for Large Scale Virtual Environments*. PhD thesis, Naval Postgraduate School, Monterey, California, June 1995. (p. 22)
- [97] Michael R. Macedonia and Michael J. Zyda. A taxonomy for networked virtual environments. *IEEE MultiMedia*, 4(1):48–56, January–March 1997. (p. 19)



- 
- [98] Robert Martin, Dirk Riehle, and Frank Buschmann, editors. *Pattern Languages of Programming Design*, volume 3. Addison Wesley, 1998. (pp. 66, 80, 181)
- [99] Sun Microsystems. Jini Architecture Specification, Version 1.1, October 2000. (p. 29)
- [100] Larry Moore. Transverse Mercator Projections and the U.S. Geological Survey Digital Products. Technical report, U.S. Geological Survey, July 1997. (p. 17)
- [101] Reagan W. Moore, Chaitanya Baru, Richard Marciano, Arcot Rajasekar, and Michael Wan. *The GRID: Blueprint for a New Computing Infrastructure*, chapter Data-Intensive Computing, pages 105–130. Morgan Kaufmann, 1999. (pp. 13, 30)
- [102] Motion Picture Expert Group (MPEG). *MPEG-4: ISO Standard 14496*. International Standards Organisation, 1999. (p. 27)
- [103] Martín A. Musicante. The Sun RPC Language Semantics. In *Proceedings of PANEL'92, XVIII Latin-American Conference on Informatics*. Universidad de Las Palmas de Gran Canaria, 1992. (p. 108)
- [104] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading MA, 1993. (p. 26)
- [105] Object Management Group. *CORBA, The Common Object Request Broker: Architecture and Specification Revision 2.0*. Object Management Group, 1995. (pp. 29, 36, 66, 76, 123, 129)
- [106] Object Management Group. *Unified Modelling Language (UML), version 1.3*. Object Management Group, 2000. (p. 65)
- [107] Orbimage. <http://www.orbimage.com>. (p. 12)
- [108] M. E. Papka and R. Stevens. UbiWorld: An environment integrating virtual reality, supercomputing and design. (pp. 2, 21)
- [109] K. Park and R. Kenyon. Effects of Network Characteristics on Human Performance in a Collaborative Virtual Environment. In *Proceedings of IEEE VR'99*, Houston, Texas, March 1999. (pp. 4, 5, 19)
- [110] R. D. Price, M. D. King, J. T. Dalton, K. S. Pedelty, P. E. Ardanuy, and M. K. Hobish. Earth Science Data for all: EOS and the EOS Data and Information System. *Photogrammetric Engineering and Remote Sensing*, 60:277–285, 1994. (p. 13)
- [111] J. Protić, M. Tomašević, and V. Milutinović. *Distributed Shared Memory: Concepts and Systems*. IEEE Press, August 1997. (pp. 20, 24, 92)
- [112] Calton Pu and Avraham Leff. Epsilon-serializability. Technical Report CUCS-054-90, University of Columbia, 1990. (p. 24)

- [113] Atul Puri and Tsuhan Chen. *Multimedia Systems, Standards and Networks*. Marcel Dekker, Inc., 2000. (p. 27)
- [114] Mary Rasmussen, Theodore P. Mason MD, Alan Millman, Ray Evenhouse, and Daniel Sandin. The Virtual Temporal Bone, a Tele-immersive Educational Environment. 2000. (p. 4)
- [115] Robert Englander. *Developing Java Beans*. O'Reilly, 1997. (p. 70)
- [116] D. J. Roberts. *A Predictive Real Time Architecture for Multi-User, Distributed, Virtual Reality*. PhD thesis, Reading University, April 1996. (p. 24)
- [117] David J. Roberts and Paul M. Sharkey. Maximising Concurrency and Scalability in a Consistent, Causal, Distributed Virtual Reality System, Whilst Minimising the Effect of Network Delays. In *Proceedings of IEEE WETICE'97*. IEEE Press, 1997. (p. 24)
- [118] Ward Rosenberry, David Kenney, and Gerry Fisher. *Understanding DCE*. O'Reilly and Associates, 1992. (p. 30)
- [119] M. Roussos, A. Johnson, J. Leigh, C. Barnes, C. Vasilakis, and T. Moher. The NICE Project: Learning Together in a Virtual World. In *Proceedings of Virtual Reality Annual International Symposium (VRAIS'98)*, pages 176–183. IEEE, March 1998. (p. 4)
- [120] D. Rus, R. Gray, and D. Kotz. Transportable Information Agents. In *International Conference on Autonomous Agents*, pages 228–236. ACM Press, February 1997. (p. 66)
- [121] Matthew D. Ryan and Paul M. Sharkey. Causal Volumes in Distributed Virtual Reality. In *Proceedings of the 1997 IEEE International Conference on Systems, Man and Cybernetics*, pages 1067–1072, 1997. (p. 23)
- [122] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications, January 1996. Internet Request for Comments 1889. (p. 27)
- [123] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol, April 1998. Internet Request for Comments 2326. (p. 27)
- [124] Silicon Graphics (SGI). IRIS Performer Programmers Guide, Chapter 10 Clip-Textures, 1997. (p. 27)
- [125] Sandeep K. Singhal and David R. Cheriton. Exploiting position history for efficient remote rendering in Networked Virtual Environments. *Presence: Teleoperators and Virtual Environments*, 4(2):169–193, 1995. (p. 23)
- [126] Sandeep K. Singhal and Michael Zyda. *Networked Virtual Environments: Design and Implementation*. ACM Press - SIGGRAPH Series. Addison Wesley, New York, 1999. (pp. 19, 163)

- 
- [127] L. Smarr and C. E. Catlett. Metacomputing. *Communications of the ACM (CACM)*, 35(6):44–52, June 1992. (p. 29)
- [128] S. E. Smith and F. W. Weingarten. Research Challenges for the Next Generation Internet. Technical report, Computing Research Association, Washington, 1997. (p. 18)
- [129] Dave Snowdon, Chris Greenhalgh, and Steve Benford. Large Scale Real Time Multi-User Virtual Reality Research (HIVE), 1997. (p. 24)
- [130] John P. Snyder. Map Projections – A Working Manual. Professional Paper 1395, U.S. Geological Survey, 1987. (p. 17)
- [131] W. Richard Stevens. *Unix Network Programming, Second Edition*, volume 1. Prentice Hall, 1998. (pp. 27, 131)
- [132] Duncan R. Stevenson, Kevin A. Smith, John P. McLaughlin, Chris J. Gunn, J. Paul Veldkamp, and Mark J. Dixon. Haptic Workbench: A multi-sensory virtual environment. In *Stereoscopic Displays and Virtual Reality Systems VI*, volume 3639 of *Proceedings of SPIE*, pages 356 – 366, 1999. (p. 153)
- [133] Sun Microsystems. *Java Remote Method Invocation Specification, Revision 1.5*. October 1998. (pp. 29, 36, 66, 123)
- [134] Sun Microsystems. *Java Advanced Imaging API White Paper*. 1999. (p. 14)
- [135] Supercomputer Computations Research Institute. *Distributed Queueing System 3.1.3 Reference Manual*. Florida State University, Tallahassee, Florida, March 1996. (p. 29)
- [136] Advanced Visualisation Systems. AVS5. <http://www.avs.com>. (pp. 14, 154)
- [137] Samuel R. A. Taylor. A Distributed Visualisation tool for Digital Terrain Models. Technical Report TR-CS-99-02, Department of Computer Science, Australian National University, Canberra, Australia, July 1999. (p. 105)
- [138] Samuel R. A. Taylor. Distributing data for Teleimmersive applications. In *Proceedings of the Sixth IDEA International Workshop*, pages 51 – 58, Rutherglen, Victoria, Australia, 1999. (pp. 18, 20)
- [139] Samuel R. A. Taylor and Brian Corrie. vGrid: An infrastructure for collaborative virtual environments. In *Proceedings of SimTecT 2000*, pages 253 – 260, Darling Harbour, Sydney, Australia, 2000. (p. 149)
- [140] Samuel R. A. Taylor and David J. Miron. Performance characteristics of a Java object request broker. Technical Report DSTO-TR-0696, Defence Science and Technology Organisation, Department of Defence, Salisbury, South Australia, Australia, July 1998. (pp. 27, 123, 130)

- [141] OpenGIS Consortium (The). OpenGIS Implementation Specifications. <http://www.opengis.org>. (p. 13)
- [142] OpenGIS Consortium (The). *OpenGIS Abstract Specification (Draft)*. 2001. <http://www.opengis.org>. (p. 13)
- [143] OpenGIS Consortium (The). *OpenGIS Coordinate Transformation Services Implementation Specification*. 2001. <http://www.opengis.org>. (p. 18)
- [144] Brian L. Tierney, Jason Lee, Brian Crowley, Mason Holding, Jeremy Hylton, and Fred L. Drake, Jr. A network-aware distributed storage cache for data-intensive environments. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 185–193, Redondo Beach, CA, August 1999. IEEE Computer Society Press. (pp. 28, 30, 41)
- [145] Henrik Tramberend. A Distributed Virtual Reality Framework. *Virtual Reality*, 1999. (pp. 19, 20, 158)
- [146] Ken Tsui, Peter A. Fletcher, and Matthew A. Hutchins. PISTON - A Scaleable Software Platform for Implementing Parallel Visualisation Algorithms. In *Proceedings of Computer Graphics International '94*, Melbourne, Australia, June 1994. (pp. 14, 17, 37, 43)
- [147] Bryan Turner. Real-Time Dynamic Level of Detail Terrain Rendering with ROAM. *Gamasutra*, April 2000. <http://www.gamasutra.com>. (p. 26)
- [148] Thatcher Ulrich. Continuous LOD Terrain Meshing Using Adaptive Quadtrees. *Gamasutra*, February 2000. <http://www.gamasutra.com>. (p. 26)
- [149] U.S. National Imagery and Mapping Association. *USIGS Geospatial and Imagery Access Services (GIAS) Specification, version 3.1*. February 1998. (pp. 13, 65, 129, 168)
- [150] Andries van Dam, Andrew S. Forsberg, David H. Laidlaw, Joseph J. LaViola, Jr., and Rosemary M. Simpson. Immersive VR for scientific visualization: A progress report. *IEEE Computer Graphics and Applications*, 20(6):26–52, November/December 2000. (p. 20)
- [151] Visigenic Software. *VisBroker for Java: Reference Manual Version 3.0*. 1997. (pp. 36, 131)
- [152] John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors. *Pattern Languages of Programming Design*, volume 2. Addison Wesley, 1996. (p. 66)
- [153] VRML Consortium Incorporated. *"Living Worlds", Making VRML 2.0 Applications Interpersonal and Interoperable – Draft 2*. International Standards Organisation (ISO), April 1997. (pp. 4, 19)

- 
- [154] VRML Consortium Incorporated. *The Virtual Reality Modeling Language (VRML) – Part 1: Functional specification and UTF-8 encoding*. International Standards Organisation (ISO), 1997. (p. 4)
- [155] VTP: The Virtual Terrain Project. <http://www.vterrain.org>. (p. 26)
- [156] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3), December 1997. Internet Request for Comments 2251. (p. 29)
- [157] C. Ware and R. Balakrishnan. Reaching for Objects in VR Displays: Lag and Frame Rate. In *ACM Transactions on Computer-Human Interaction*, volume 1, pages 331 – 356, December 1994. (p. 19)
- [158] R. Waters, D. Anderson, J. Barrus, D. Brogan, M. Casey, S. McKeown, T. Nitta, I. Sterns, and W. Yerazunis. Diamond park and spline: Social virtual reality with 3D animation, spoken interaction and runtime extendability. *Presence*, 6(4):461–481, 1997. (p. 22)
- [159] Kent Watsen and Mike Zyda. Bamboo – A Portable System for Dynamically Extensible, Real-time, Networked Virtual Environments. In *Proceedings of Virtual Reality Annual International Symposium (VRAIS'98)*. IEEE, March 1998. (pp. 19, 66)
- [160] Kent Watsen and Mike Zyda. Bamboo – Supporting Dynamic Protocols for Virtual Environments. In *Proceedings of IMAGE'98*, Scottsdale, Arizona, August 1998. (p. 66)
- [161] Drew Whitehouse. Building Screen Based Immersive Virtual Environments on a Budget - the Wedge. *ACM SIGGRAPH Computer Graphics*, 33(4), 1999. (p. 153)
- [162] Matthew D. Wilson, Samuel R. A. Taylor, Michael Rezny, Markus Buchhorn, and Andrew L. Wendelborn. ACSys/RDN experiences with Telstra's Experimental Broadband Network. Technical Report TR-CS-98-03, Department of Computer Science, Australian National University, Canberra, Australia, April 1998. (p. 106)
- [163] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 1999. (p. 153)
- [164] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0, Second Edition*. October 2000. (p. 53)
- [165] K.-L. Wu, P. S. Yu, and C. Pu. Divergence control for epsilon-serializability. In *International Conference on Data Engineering*, pages 506–517, Los Alamitos, Ca., USA, February 1992. IEEE Computer Society Press. (p. 24)



---

# Index

---

- Comet, *see* Experimental systems, Comet  
CROP, *see* Experimental systems, CROP
- Design Patterns
- Abstract Factory, 49, 77, 166–168, 172, 174
  - Accepter and Connector, 78, 169
  - Active Object, 64
  - Adaptor, 124, 147
  - Facade, 75, 136, 172
  - Master-Slave, 44, 89, 90, 92, 93, 182, 184
  - Master-Worker, 44
  - Model-View-Controller, 19
  - Observer, 81, 172, 174
  - Proxy, 92, 182
  - Reactor, 81
  - Serializer, 64, 179
  - Streams, 37
- Design Patterns, 9
- Experimental systems
- Comet, 9, 103–111, 113, 115, 119, 127, 143, 145, 152, 156, 160
  - CROP, 104, 121, 137–142, 144, 160
  - IMAD, 121, 126–128, 131, 133–143, 160
  - OATS, 121, 122, 124, 126–128, 139, 160, 161
  - vGrid, 9, 74, 104, 119, 142, 143, 145–157, 160
- Geospatial imagery, *see* Satellites
- IMAD, *see* Experimental systems, IMAD
- Implementation profile, 63, 64, 101, 149, 160
- Encoding the resource schema in a directory hierarchy, 74, 155
  - Enumeration of Operator types, 68, 72
  - Enumeration of Pipeline types, 72
  - Factory service properties, 68
  - Names of external interfaces to factories, 71
  - Pipeline trader interface, 66
  - Priority values for tile requests, 84
  - Protocol-specific connection details, 85
  - Request mapping functions, 85
  - Scheduling policies used in the RACE, 99
  - Serialisation mechanisms for mobile objects, 84
  - Tiling and Level of Detail mechanisms, 96, 101
  - Usage policies for Specialist Networks, 73
- OATS, *see* Experimental systems, OATS
- RAPID classes: dissemination pipeline
- Accepter, 77–79, 84, 170–174, 178, 182–184
  - Approximator, 78, 79, 86–88, 90, 92, 93, 96, 109, 170, 171, 173, 175, 176, 183–185
  - Connection, 76–79, 82, 88, 91, 96, 124, 126, 131, 139, 141, 167, 170, 171, 173–175, 178, 183, 184
  - DependentSet, 87–89, 91, 171, 173, 175, 183, 185
  - InputPort, 75, 77, 88, 92, 96, 170–175, 183, 184
  - Link, 77–79, 81, 82, 88, 90–92, 96, 97, 107, 131, 170–175, 177–185
  - LinkFactory, 77, 78, 170, 171, 174

- 
- MessageListener, 82
  - Operator, 65, 66, 75, 76, 80, 96, 129, 165–169, 171–173, 175, 182
  - OperatorPort, 75, 79, 81, 82, 85, 171–175, 177–185
  - OutputPort, 75, 77, 82, 88, 92, 96, 97, 101, 170, 172–174, 183, 185
  - OutputQueue, 87, 89–91, 173, 175, 176, 183
  - OutstandingSet, 86–89, 91, 97, 98, 171, 173–175, 183, 185
  - RequestHandler, 79, 81, 82, 86–90, 92, 98, 171–185
  - RequestMap, 78, 82, 88, 91–93, 96, 107, 170, 171, 175, 179, 181, 183, 184
  - SupportedQueue, 87–90, 92, 175, 176, 183
  - TileBuffer, 77, 79, 85, 88–90, 92, 93, 96–98, 109, 170–175, 181–185
  - Transformation, 79, 87–94, 96, 97, 170, 172, 173, 175, 176, 183
  - RAPID classes: application management
    - Filter, 66, 134, 165, 167, 168, 172
    - FilterFactory, 66, 134, 139, 165, 168
    - ImageAccessor, 66, 69, 134, 140, 165–167, 172, 181
    - ImageryArchive, 66, 69, 134, 147, 165, 166, 168
    - OperatorFactory, 65, 66, 69, 72, 165–169, 172
    - PipelineManager, 65, 66, 68, 72, 74, 77, 100, 134, 136, 137, 147, 157, 165–169, 171–173, 182, 183
    - PipelineTrader, 66, 68, 72, 134, 140, 151, 165–169
    - ServiceInstance, 165, 166, 168, 169, 172
    - ServiceNegotiation, 65, 68, 69, 166–169
    - ServiceProvider, 165–169
    - ServiceTicket, 68, 75, 165–169
    - Transporter, 66, 71, 73, 167, 169, 172
    - TransportFactory, 66, 71–73, 168, 169
  - RAPID classes: RACE
    - BroadcastLink, 99, 174, 177, 178, 180, 182, 183
    - BroadcastPort, 96, 99, 172, 177, 178, 180, 182, 183
    - RACEFactory, 66, 72, 166–168, 182, 184
    - RACEHandler, 94, 96–98, 174, 182–185
    - RACEPort, 96, 171–173, 182, 183
    - RACESlavePort, 96, 97, 171–173, 183–185
    - Scheduler, 94, 98, 182–184
    - Slave, 93, 94, 96, 97, 182–185
    - SlaveHandler, 96, 97, 174, 183–185
    - SlaveInputPort, 171, 172, 183, 184
    - SlaveOutputPort, 172, 173, 183, 185
  - RAPID classes: RAPPORT
    - CancelRequest, 84, 98, 177, 179, 180
    - CausalMessage, 99, 177, 179
    - CloseLink, 85, 177, 179, 180
    - CollaborativeMessage, 99, 177, 179, 180, 182
    - FlowControl, 85, 98, 175, 178–180, 184
    - IncreaseFlow, 85, 178–180, 184
    - LimitFlow, 85, 178–180, 184
    - MapUpdate, 81, 85, 88, 97, 175, 179, 180
    - Message, 82, 84, 176–181
    - NonCausalMessage, 99, 177, 179, 180
    - OpenLink, 78, 84, 85, 170, 174, 178–180
    - PipelineMessage, 84, 176–181, 185
    - TileMessage, 84, 175, 176, 179–181
    - TileRequest, 81, 84, 92, 97, 99, 101, 171, 173, 175, 177, 179–181, 185
    - TileResponse, 82, 84, 85, 88–92, 97, 179–181
  - Requirements



- 
- Access throughput, 5, 8, 9, 27, 28, 31, 33–35, 38, 43, 59–62, 85, 93, 98, 101, 119, 121, 122, 127, 131, 134, 137, 138, 140–143, 159, 160, 184
  - Application management, 5–9, 27, 29, 30, 33, 34, 48, 59–62, 65, 101, 119, 122, 127, 131, 134, 137–146, 155, 156, 159, 160, 168
  - Client responsiveness, 5, 8, 18, 27, 28, 31, 33–35, 38, 43, 47, 59–62, 93, 98, 101, 103, 104, 110, 111, 119, 124, 126, 127, 131, 137, 139, 143, 145, 159–161, 184
  - Collaborative data sharing, 5, 6, 8, 18, 23, 31, 33, 41, 59–62, 85, 93, 94, 96, 100, 101, 127, 131, 139, 159–161, 182
  - Satellites
    - GMS-5, 2, 121, 122, 124, 125
    - Ikonos, 12
    - IRS-1, 3, 53, 104, 108, 119
    - Landsat, 2, 3
    - NOAA, 14
    - OrbView, 12
    - SPOT, 2, 3
  - Techniques
    - Approximating tiles, 8, 47, 79, 86, 88, 159
    - Asynchronous communication, 159, 160
    - Connection object, 8, 35, 36, 45, 56, 64, 77, 78, 89, 96, 140, 142, 156, 170
    - Data sharing with causal order, 24, 100, 177, 180
    - Inter-site sharing bus, 8, 41, 42, 58–60, 94, 97–99, 101, 160, 177, 180, 182
    - Level of detail, 24, 122
    - Life-cycle of pipelines, 8, 49, 51, 52, 60, 61, 65, 66, 74, 79, 160, 166, 167
    - Parallel processing, 61, 79, 159, 184
    - Parallel streaming, 8, 35, 43, 45, 60, 61, 74, 79, 82, 84, 89, 96, 107, 159, 160, 171, 175, 179, 181
    - Pipeline scheduling, 8, 38, 41, 60, 94, 113, 119, 122, 126, 159, 160, 178, 179, 184
      - by Area of Interest, 39, 58
      - by Level-of-Detail, 39, 58
      - by Motion Prediction, 40, 42, 58
      - by Request Proximity, 39, 58, 90, 106, 110, 113, 119, 161
    - Request map, 82, 107
    - Resource schema, 8, 49, 50, 60, 66, 160, 168
    - Service tickets, 52, 60, 160
    - Site cache, 8, 40, 44, 58, 60, 92, 93, 106, 122, 126, 159, 160, 182, 184
    - Tiling, 77, 85
  - vGrid, *see* Experimental systems, vGrid



---

# Producing a Thesis in 2001

---

We live in a time of great change, and work in an industry which drives much of this change. As a technologist it's a constant source of amusement to reflect on the hardware and software I used only a few years earlier. So here, for the sake of posterity, are the tools used to produce this thesis.

My office desktop machine was a generic PC running Windows NT4 on a 366MHz Intel Celeron CPU, 256MB RAM, a 4GB hard disk, and a 16MB nVidia TNT graphics card driving a 21" CRT Monitor at 1152x864 @ 100Hz. Away from the office I also used a Digital HiNote 2000 notebook, which ran Windows 98 on a 166MHz Intel Pentium (P5) with 64MB RAM, a 3GB hard disk and a 14.1" TFT LCD with a resolution of 1024x768 – slim and very gentle on the eyes but it weighed a ton and boy did it give off some heat.

The thesis was typeset in  $\LaTeX$  with the bibliography kept in BibTeX. Almost the entire text was written out by hand first, transcribed into Microsoft Word 2000 then converted to text files, and marked up in  $\LaTeX$  using one of two text editors: Emacs 20.5.1 or TextPad 4.3. This convoluted process and eclectic choice of tools drove my fellow students into fits of confusion and righteous indignation – which I secretly reveled in. Diagrams and support material were included as Encapsulate Postscript files produced by Microsoft Visio 2000, Corel Draw 9 and Jasc Paint Shop Pro 6. This worked well for the structured drawings, but was very inefficient for the large bitmaps.

Two things kept me sane through the long years of struggle. First there was the music: an 18GB archive of MP3 tracks kept on a local Linux server, and streamed by HTTP over a 100mbps Ethernet network. I doubt I would have made it without the help of Gomez, Radiohead, Powderfinger, Muse, Travis, Fat Boy Slim, Moby and all the fantastic Triple-J Hottest 100 compilations. Second there was one game above all others: Age of Empires developed by Ensemble Studios and published by Microsoft. Every version, every update, John, Zhen and I played them all. It was simply the perfect multiplayer LAN game for us. Nothing matched the satisfaction of beating three or four computers on the hardest settings: John's unassailable squads of English archers, Zhen's great flocks of Paladins (often sitting idle), my plucky Teutons or Mayans plugging the gaps. Great game, great fun, great mates.