DOCTOR OF PHILOSOPHY

Real-Time and Interactive Computer Graphics in Grid Environments

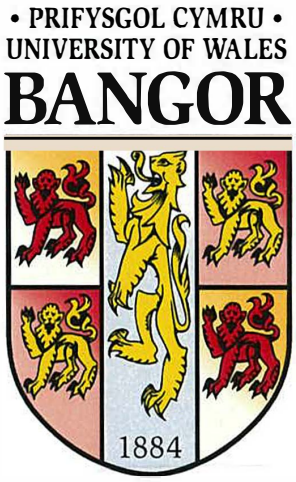
Fewings, Ade

Award date:
2006

Awarding institution:
University of Wales, Bangor

Real-Time and Interactive Computer Graphics in Grid Environments

Adrian J Fewings



Thesis Submitted in Candidature for the Degree of
Doctor of Philosophy

February 2006

School of Informatics
University of Wales, Bangor
United Kingdom

Abstract

The change in developed society brought about by the computer revolution shows no sign of abating. The daily utilization of computers and their communication capabilities has altered the way people communicate with each other and organise their lives. As more and more of the world around us become reliant on technology, so we are faced with the challenge of continuing the development of services and ideas whilst managing and adapting existing facilities.

Computer visualization provides hidden perspective and depth to users. Computer graphics technology that was, not long ago, hugely expensive now enables anybody to sit at home and experience a real-time, interactive virtual world of high detail and reality. As occurred with computation some years ago, the desire for ever more power has led to parallelisation of graphics processing, enabling ever more extraordinary graphics to be produced in real-time. In line with visualization, the development and spread of networked computing enables the world to work in fundamentally different ways. We are rapidly transiting to a world of immense connectivity in which the challenge is to offer new and existing services in an accessible manner. This is the world of Grid computing.

We have developed a system, called *jgViz*, that provides an end-to-end user experience in real-time, interactive visualization in the Grid world. This is an area which is often overlooked by other projects in Grid visualization, which have so far focused on non-interactive scientific visualization. *jgViz* uses the Grid information services and runtime components of the Globus Toolkit, alongside the Chromium parallel graphics system to enable existing standards-based applications to be used in, and gain from, the Grid environment.

The information model we have designed registers and advertises the presence and capabilities of available jgViz servers through an efficient syntax in a number of Meta-Directory Service (MDS) servers. Our client implementation discovers resources through this hierarchy and communicates with the jgViz servers themselves in order to learn updated statistics on server status in terms of loading and network latency. This information is utilised by the scheduling system of the client to construct a parallel graphics pipeline on available resources, to suit a user's high-level requirements. The pipeline created is scheduled to provide best performance in either a sort-first tiled-wall or sort-first readback configuration. To launch a pipeline, we utilize the Grid Resource Allocation Manager (GRAM) protocol for job submission, and the GridFTP and Global Access to Secondary Storage (GASS) protocols for data transfer. Once a pipeline is up-and-running, jgViz will monitor it in order to detect when external factors reduce pipeline performance. When such an event occurs, jgViz will stop the pipeline, reschedule it among the available jgViz resources and restart it.

We have carried out an in-depth study of the performance of the jgViz system, in terms of jgViz-specific capabilities and the limiting factors of such distributed graphics pipelines. We find several things: network performance is the most critical factor with Fast Ethernet not offering acceptable jgViz performance and network latency being key; the 'double-hit'

of latency alongside slow readback performance in a readback pipeline restricts the usability of such setups; and that jgViz monitoring does not interfere with the usual operation of a pipeline but does trigger when pipeline performance is impinged in some way. Given these limitations, we conclude that a general-purpose, real-time Graphics pipeline can be provisioned using Grid technologies.

Contents

List of Figures

Acknowledgements

This thesis written to the tunes of lots of music, the most reasonable (by my judgement) stuff among which is as follows: The Cardigans, Flaming Lips, The Simpsons, Norah Jones, Nightmares on Wax, Siobhan Donaghy, Metallica, Emiliana Torrini, Natalie Imbruglia, Faithless, Goldfrapp, KT Tunstall, Smoove, Husky Rescue, Mylo, Zero 7

Thanks is due to the following people for the following things (in no particular order):

- Rob for increasing my musical exposure during the last few years.
- The two Davids in ITS' Networks for the Cat loan, Terence for the GeForce loan, Matthew and John in Informatics for r319 loan and commandeering over the summer.
- Prof Alan Shore for understanding the difficulties that have been encountered.
- Dr Gareth Roberts for being a realist at an important time.
- Rob and Les for assistance setting up 'Informatics Cabling Inc.'
- Rob and Matthew for the consumption of some disk and bandwidth for paranoia-driven backups.
- All in ITS (particularly the bosses, Sim and Julie) for being more than decent with me whilst I've been writing this thesis.
- The e-Viz project team - (Manchester - Mark & John), (Leeds - Jason & Ken), (Swansea - Chen, Mark, David & Nicolas) - it was a pleasure to experience the early stages of the project and learn so much from you all.
- Rob and Nigel for proof-reading and battling through my dodgy English.
- Franck and Chris for so many UT CTF sessions, good times and the pleasure of being in the same group as you.
- Nigel for rescuing things from a bad situation. Two years ago, I didn't think I could or would be here - massive thanks are due for the supervision and for showing me what the PhD process can really be. 'Indebted' probably doesn't do it justice.
- My colleagues in the Blaise Pascal Lab for the last four years - Nidal, John, Les, Rob - beyond everything that has happened in the last four years, I will always fondly remember my time in the lab. It's been interesting, it's been fun - deeply felt thanks to you all.
- Donald, for always making things not seem so bad.
- Mike 'Sandwich' for lunches, laughs and occasional reality checks.
- Chris H for the moral support, encouragement, listening to the tails of woe, and pleasantly side-tracking conversations.
- My family, for support and for all those years before and during Bangor. I'm glad you taught me to work hard.

Document Information

This thesis was prepared using JabRef and Kate in KDE on Solaris Sparc. The thesis was compiled using LaTeX 2_ε.

"You see, wire telegraph is a kind of a very, very long cat. You pull his tail in New York and his head is meowing in Los Angeles. Do you understand this? And radio operates exactly the same way: you send signals here, they receive them there. The only difference is that there is no cat."

Albert Einstein (1879 - 1955), describing radio [or could it have been the Grid?]

Chapter 1

Introduction

1.1 Introduction

The last fifteen years has seen a technology revolution in developed society. The extraordinary growth in computing, fueled by the emergence of mass-connectivity in the form of the Internet, has fundamentally changed mankind. Where computers were once large, loud, expensive and only to be found doing specialized work or scientific computation, most people could not now imagine a world without daily computer interaction for all sorts of purposes, including work and entertainment. Where once the only way to communicate with a friend or family member a long way away was to make an expensive phone call or post a high-latency letter, now emails are delivered in seconds and instant-messaging and Voice-over-IP applications make involved communication cheap.

However, this is only the beginning. We now carry around devices (or a single device) in our pockets that allow us to play games, hold thousands of songs, have a face-to-face conversation with someone on the other side of the world, connect to computers the world-over and enable other people to track our movements. More and more of the things we interact with on a daily basis are connected to this 'network' - from microwaves that download recipes to Radio Frequency Identification (RFID) based supply-chain management systems. Computers continue to enable our imaginations through the visualization of fact and fiction with immensely powerful graphics systems.

We are now faced with a new challenge. Computing is part of everyday life, yet, computing continues to develop. How do we manage this development for users whilst allowing the

development to take place? As we go forward, there is a need to adapt to these new technologies but still be able to retain abilities from one perceived generation of computing to the next. This thesis is concerned with exactly that.

1.2 Thesis Motivation and Focus

Computer graphics is a field that has come from humble beginnings in computer science to mainstream use in entertainment, education and research. The ability to visualize immensely complex structures and ideas can enable mankind to go beyond the limits of imagination, both in terms of content and timeliness. Real-time graphics has become particularly popular in recent years, primarily through the development of powerful commodity graphics accelerators as used today in the full range of products from supercomputers to games consoles. As parallel computing was to the processors available and requirement for computation ten or twenty years ago, so parallel graphics is to the graphics accelerators and desire to visualize today. Parallel graphics processing offers truly unique opportunities to engage in real-time visualization of extraordinary worlds and models, through the coupling of multiple graphics accelerators. In the same way that high-performance computation has become more distributed, so the same is happening with high-performance visualization and graphics. Increasingly capable networks allow a real-time visualization to occur across a myriad of machines that may be geographically dispersed. Key work from a number of research groups has built this research area up and it continues to grow - see Chapter 2 for details.

The breathtaking development of networked computing in recent years has created a world in which goals are achieved in radically changed ways. Looking further forward, developments in areas such as IPv6 and wireless networks will create an even more connected world in which the majority of electrical items take advantage of the network. The combination of this near fully-connected world and the service infrastructure to support it is what the next generation is all about. As the first generation of this connected world was widely known as the Internet and the second as the Web, so the third is the Grid. Grid research began in the early to mid 1990s as networking technologies matured enough to allow the efficient and economic transfer of data across larger distances. This enabled compute clusters at distributed sites to be shared. The early attempts to implement software to do this gave rise to several key problem areas, including security, resource allocation and platform heterogeneity, that define the very concept of Grid computing. The work of the Globus team was instrumental in beginning to tackle these problems as the Globus Toolkit

was developed towards providing an infrastructure for high-performance computing. Major developments facilitated by the Globus project tackled providing an information exchange system and a distributed file system, both to support a supercomputing infrastructure. In the United Kingdom, the National Grid Service [1] is a production Grid service that grew out of the prototype UK e-Science Grid and provides computational and data Grid abilities. *e-Science* is the term used to represent the global sharing and collaboration required to tackle new problems in science and engineering. The e-Science programme was launched in November 2000 as a cross-Research Council funded initiative to develop solutions and middleware for enabling e-Science and forming a base for future commercial e-business.

A number of projects have brought Visualization and Grid research together. Many have been aimed at either specialized visualizations or taking advantage of abilities offered only by Grids, due to their scale and the high-performance facilities involved. There is, however, a need to enable existing standards-based applications to take advantage of the Grid. OpenGL is a very widely used graphics standard primarily intended for interactive and real-time rendering. As it represents a 'gap in the market' this thesis is aimed at taking existing OpenGL applications and providing an environment for them to run with the advantage of Grid facilities. We introduced this Grid visualization system first at SuperComputing 2004 [3] and call it *jgViz*.

1.3 Hypothesis

The combination of Visualization and the Grid leads to many questions. We surmise these questions, facts and aims with our thesis hypothesis. The jgViz project is an investigation of this hypothesis.

Through the use of indexing systems, job scheduling and data transport mechanisms, the Computational Grid can be used to provision a general-purpose, real-time, distributed graphics pipeline.

[1] See `www.ngs.ac.uk`

1.4 Contribution

The major contribution of this work is in the design and implementation of a Grid visualization system that enables existing standards-based applications to function in, and benefit from, the Grid environment. Through the use of Grid information services, Grid runtime components and an existing non-Grid parallel graphics system, we produce a unique discovery, scheduling and monitoring solution for Grid-based visualization.

1.5 Thesis Structure

In order to position this work within the scope of that which has led to it, a review of key technologies and developments in presented in chapter 2. The development of Grid computing is discussed and how this moves through a number of projects and generations to became modern Grid research. Combining this with the concepts of visualization and high-performance graphics, we conclude with a review of existing Grid Visualization work.

Chapter 3 begins our discussion of our implemented software, jgViz. There are a number of component parts that go to make up jgViz and in this chapter we discuss that which underpins all others - the jgViz Grid Information Model. We discuss the difficulties associated with developing a language to describe visualization and explain the basis of the language we have developed. We discuss the Grid server and client aspects of how a discoverable resource is made available and provide an example to demonstrate the information path throughout the jgViz experience.

After information has been learned and resources discovered, the next task is to apply that information, as we discuss in Chapter 4 on Scheduling. Covering the multi-stage scheduling process that goes about determining the resources to be used by jgViz, this chapter discusses the scoring system we use to determine what makes one node more appropriate for the task than another. We also describe the process by which the scheduled data is converted into a collection of Grid resources that utilise the Chromium parallel graphics system.

Once resources have been chosen, chapter 5 discusses the pre- and post-launch elements involved in creating an actual pipeline by utilising Grid protocols for job launching and control. We cover the mechanisms required for launching jobs and then compare the scalability of round-robin and concurrent systems for monitoring a number of nodes. We also deliver explanation of how this fits in with jgViz deciding that a running graphics

pipeline has suffered a performance problem, and on the events that such a trigger causes. We conclude with a discussion on further ways to monitor nodes and the implication thereof.

Chapter 6 presents a study of the performance of distributed graphics pipelines in the jgViz system. We focus not on the absolute performance of the pipeline, but primarily on the monitoring and rescheduling features that jgViz provides in order to increase the reliability of such pipelines. We look at both high and low level performance metrics in idealised and congested networks and observe the patterns of traffic, the effects of different node types and the overall scalability of the system. We also give consideration to optimization of the customizable settings within jgViz to improve the end-user view of a distributed graphics pipeline. This performance study allows us to develop understanding of the limitations and speculate on possible technical developments, with regard to jgViz.

Finally, in chapter 7 we present conclusions from the jgViz body of work and discuss future work. We determine whether or not our hypothesis has been proved.

1.6 Publications

Work from this project has been accepted for publication as follows:

- *Visual Supercomputing - Technologies, Applications and Challenges.* K A Brodlie, J M Brooke, M Chen, D Chisnall, A J Fewings, C J Hughes, N W John, M W Jones, M Riding and N Roard. STAR - State of the Art Report, Eurographics 2004.
- *Distributed Graphics Pipelines on the Grid.* A J Fewings and N W John. Poster presentation, Supercomputing 2004.

Chapter 2

Background and Literature Review

There are two principle objectives of this chapter - firstly to give the background to the research, and secondly to report on the literature research undertaken. The scope of the project has largely been covered in the previous chapter, but it is important to fill in details on the processes that brought this work to life. The literature review consists of a search for information on the main contributory technologies to this work. A definite path is followed through this to build up all the relevant sections to an end-product providing the knowledge necessary to see from where and how this thesis has evolved.

2.1 High Performance Visualization

Computer graphics and visualization has always used high performance computing facilities. *High Performance Visualization* (HPV) is the term used to refer to the latest visualization techniques being used with *High Performance Computing* (HPC) resources. HPV tasks thus normally involve large, high resolution datasets and computationally intensive tasks. The use of networks, immersive environments and high resolution graphics facilities really makes HPV what it is today. It is, perhaps, best summarised by this description from the EPSRC e-Viz project proposal: 'A typical HPV task is a complex feedback process, involving data collection, visualization design, task parallelisation, immersive visual display and interfacing with the corresponding data generator such as a simulation engine' [1]. This thesis is concerned with a particular application of parallel computing techniques within High Performance Visualization (HPV).

[1]The e-Viz project can be found on the web at `www.eviz.org`

2.1.1 Parallel Visualization

Parallel visualization is broadly divisible into two fundamental types - Object Space and Image Space [4]. Object Space parallel refers to the decomposition of the visualization task into separate parallel parts being carried out on the actual dataset involved. Image Space parallel refers to the decomposition taking place on the basis of coordinate space in the resulting image.

Object Space

Within Object Space parallel visualization, each node (individual unit of parallel system, typically a single machine or processor) is responsible for the rendering of its block of data, irrespective of whether it may actually be visible at that precise moment. Object Space parallelization is also known as Sort-Last [5], reflecting the late stage in the graphics pipeline at which the graphics primitives are sorted from object-space into the resultant image-space (See Figure 2.1). There are three principal data partitioning schemes in Object Space parallelism.

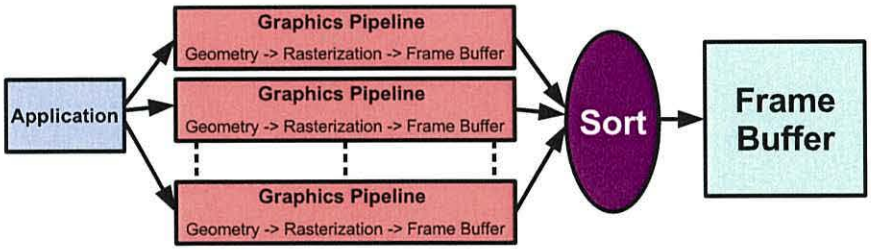


Figure 2.1: Sort-Last Graphics Pipeline

Firstly, there is the concept of Complete Data Replication in which each processor holds all data locally. This allows simple parallelization through the same sequential algorithm on all nodes and also reduces any requirement for communication during processing. However, the memory overhead is significant and this technique does not scale well. For example, as a data set grows then each node requires more memory to hold it and, as the parallel machine size grows, then the cost of initial data distribution soon leads to this approach becoming impractical. This approach is also useful for read-only data due to the excessive cost in time and processing of modifying the data on one node and the necessary redistribution to all nodes in a consistent manner.

Secondly, Object Space data partitioning can be based on sets of object data without structure. Such a scheme is called Non-Hierarchical and can be of a regular or irregular form. In both of these, partitioning is typically carried out on a block or slice basis. Regular Block Decomposition breaks the 3D data set into equal-sized regular blocks whereas Irregular Block Decomposition produces unequal sized blocks. The regular form suffers from the potential downfall of not being well load-balanced whereas the irregular form deliberately produces blocks that contain similar amounts of work. However, the overhead of this can be significant as it can be an intensive pre-processing task. Regular Slice Decomposition produces consistent width, full-length slices of the data set in contrast to the differing widths produced by Irregular Slice Decomposition. Producing a sliced decomposition can give the added advantage of neighbouring nodes processing neighbouring slices, which can be advantageous for data exchange in certain single-layer network structures.

Thirdly, some form of multi-layer structure can be used for the partitioning of data in the shape of a Hierarchical Partitioning scheme. The Kd Tree Partitioning approach is an example of this. Kd trees [6] are used for partitioning from a k-dimensional space into subvolumes along planes through the dataset. Octree Subdivision [7] [8] represents a recursive decomposition of object-space into cubes of integer powers-of-2 in size. In most cases this results in a largely equal partitioning, subject to an even distribution of objects in the object space [9]. Hence, it is particularly useful with regard to partitioning of a single object, such as a volume dataset.

Image Space

In Image Space parallel visualization, the parallelism is achieved in the resultant visualization space. Traditionally, this means that different pixels in the resultant image are computed in parallel according to some scheme. Image Space parallelization is also known as Sort-First, describing the sorting between object space and resultant image space occurring early in the graphics pipe (See Figure 2.2).

A basic scheme is Regular Image Space parallelism, in which the individual pixels of the required output (considering display device and resolution) to be produced are placed in a queue and processed in a farm parallel manner, producing a good load balancing system with no impact if a particular pixel takes a substantially longer time to render than others. However, the extremely fine grained parallelism that this represents has a substantial communication overhead, therefore restricting the types of parallel machine that can be used. If the granularity is increased by parallelizing on a scanline or pixel group basis,

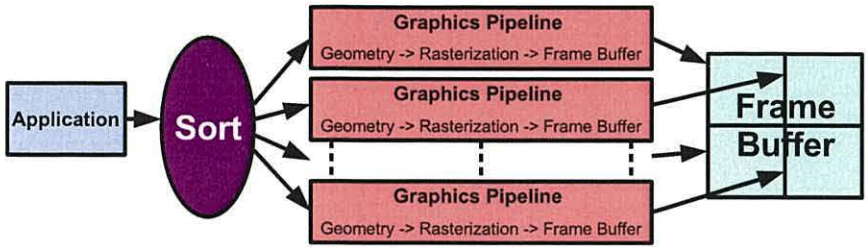


Figure 2.2: Sort-First Graphics Pipeline

the communication overhead becomes less significant. In such a system, the scanlines are distributed in a round-robin fashion, thus providing some inbuilt load-balancing as some pixels on a scanline will take longer than others to render. The granularity can be further reduced by grouping neighbouring scanlines together to a certain block size. Such grouping techniques can also be applied to create blocks of pixels in a regularly spaced tiled arrangement. This exploits the data coherence between neighbouring pixels without a substantial cost for block-to-block communication and allows the size of the blocks to be tuned to suit the granularity of the machine. If some pre-processing is added into this, then irregularly spaced blocks can be used, improving load-balancing in exchange for the relatively small initial cost. By subsampling the image plane, the pre-processing stage may cluster small tiles together to form load-balanced larger tiles. An alternative irregular approach is for each processor to render a block of small tiles held in its queue and when it finishes, to 'steal' tiles from other processors that are still busy. Stompel et al [10] efficiently load balance in their compositing algorithm on the basis of two factors. Firstly, the data is partitioned so that each processor is responsible for data throughout the entire viewport. Secondly, the processed pixels are classified as one of three type: Background Pixels that can be ignored; Non overlapping Pixels that can be directly delivered to the display host; and Overlapping Pixels that require actual computational compositing.

Sort-First parallel rendering requires a final stage to bring together the parallel streams to form the view for the user in a single view. This will often be in the form of a compositing stage that merges the view section from the parallel render nodes into a single view. This requires inter-process communication that can often be very data intensive, subject to scene content. This is a significant performance issue for parallel and distributed visualization, depending on how the parallel pipelines are connected to the final frame buffer (for example, four graphics pipelines in one machine share superior connections than if these components are on separate machines connected only by the network). There has been various work on producing both software and hardware to optimize the compositing process and there are

many examples that could be cited. Two particularly good examples are: Ma et al [11], with their new algorithm that load balances the compositing process as much as possible by compositing all rendering node output simultaneously rather than sequentially; and Yang et al [12], with their look at three algorithm variations; both of which involve the binary swap procedure. Hardware compositing devices are effectively networked framebuffers and are somewhat rarer, but Lightning-2 [13] is a prime example. Muraki et al [14] present particularly interesting work on parallel volume visualization compositing using binary-tree structure to enhance scalability.

2.1.2 Cluster Visualization

Visualization is considered to be an effective method through which to gain understanding and insight of otherwise confusing, incomprehensible data sets. However, traditional high-end visualization systems present a number of problems, including high-cost due to their proprietary nature, restricted scope for upgrades and limited scalability. In recent years, there has been an explosive birth and growth of commodity graphics accelerators in both games consoles and workstations, motivated primarily by the gaming community. Also, a number of new networking technologies have become available facilitating both high bandwidth and low latency for relatively modest costs. The massive industry investment in all of these near commodity cost components represents an opportunity to produce a high-end visualization system that no longer suffers from the disadvantages mentioned above. By building visualization clusters, we can harness off-the-shelf components, remove the proprietary overhead and also achieve scalable and economically upgradable performance. The current trends in performance growth will not continue forever, however. As the physical limitations of silicon are increasingly reached (or, at least, progress is slower), parallelism becomes increasingly necessary in order to gain improved throughput. This has been seen happening in the processor market recently, and also the graphics processor market shows similar signs. In the future, increasing the parallelism will be the way to improve throughput. Further afield and as cheap as equipment may become, this implies that distributed parallelism and resource sharing (for example, within a Grid environment) will be an effective way to increase performance.

Chromium

Cluster rendering systems have typically been constructed as specific algorithms for tackling specific problems. Chromium is a general-purpose system for enabling cluster visualization through a stream processing architecture. As the open-source successor to WireGL [15], Chromium enables sort-first, sort-last and hybrid parallel rendering. Generality is achieved in Chromium through the use of OpenGL as the underlying language. This enables Chromium to run existing applications with either little or no modification as well as running customized Chromium applications that utilise the programmable architecture to provide different algorithmic architectures.

A Chromium rendering pipeline is configured as a directed acyclic graph (DAG) of nodes, where a node is an instance of the Chromium runtime. Generally, therefore, one node is one machine in a cluster. There are two types of Chromium runtime nodes. Firstly, a client node hosts applications and either tricks those that use OpenGL in to using Chromium's libGL instead of the system libGL (by preloading the Chromium library using the runtime linker environment variable LD_PRELOAD), or hosts directly those that are customized. Secondly, a server node is one that receives GL command streams and processes them. This can involve rendering and resolution of ordering issues. The packing, unpacking and transmission of GL commands between nodes is handled as part of the DAG. The DAG itself is represented in Python script in which nodes are created and have Stream Processing Units (SPUs) allocated. SPUs are the basis of the modular Chromium framework. They are connected by pipes over which they 'converse' in OpenGL commands to handle requirements in rendering, processing and communication. SPUs are implemented in a hierarchy and all extend from the 'passthrough' base class, which takes in a GL stream and passes it out unmodified. Example SPUs supplied with Chromium include: 'tilesort' that divides up a scene on a grid basis, 'render' that renders an OpenGL stream in a window using locally available facilities, and 'pack' that packages GL commands for transmission across a network to other nodes. Through the use of the various SPUs, Chromium can achieve the different sorting classifications as discussed above. The launching of a pipeline's components is not a trivial task, due to the potential complexity of configurations amongst so many components. Chromium solves this by having all pipelines components connect to a 'mothership' additional node to discover their configuration (e.g. what SPUs are required and where stream data comes from and goes to) as their first task upon execution, hence the only configuration required is the name of the machine that is running the Chromium mothership. The mothership itself is a simple server that contains the pipeline configuration in a single file and returns to the correctly named machines their relevant data to inform

them of their function.

Figure 2.3 shows a basic sort-first tiled display configuration, in which a single application is run through a 'tilesort' that splits up the view to be rendered by the slave servers, each of which has the relevant screen in the tiled mural attached to it and uses the local graphics hardware for maximum performance. The potential bottlenecks within this configuration are twofold. Firstly, the network between the client node and the slave server nodes needs to be of high enough performance to transport the data twixt the two without limitation. Secondly, the 'tilesort' can be quite a costly process when carried out on relatively complex scenes and so requires hardware of capable performance.

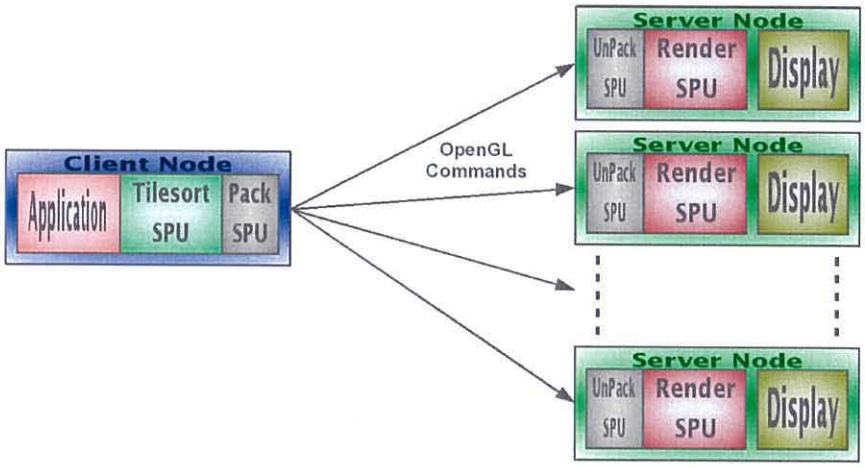


Figure 2.3: Chromium Basic Sort-First DAG Configuration

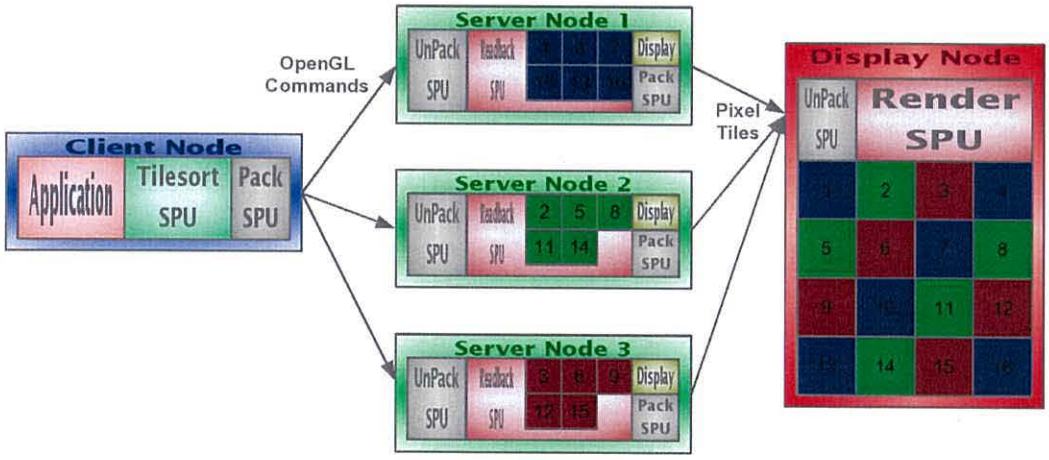


Figure 2.4: Chromium Hybrid Sort-First for Advanced Tiling DAG Configuration

Figure 2.4 shows an advanced sort-first tiling configuration, in which a single application's

graphics stream is split into smaller tiles before being fed to a number of servers. However, these servers now render the relevant tiles of data they have been assigned into their frame buffers and then issue a *glReadPixels* command to extract the 2D output. This is the effect of the 'readback' SPU. The result is then sent over the network to a server with a single screen attached that is responsible for compositing the tiles back together as appropriate and displaying them. As the pixels are sent over the network as GL commands, this reassembly process is actually a 'render' SPU. The smaller tiles used in this scheme lead to a more load-balanced offering across the available readback servers, for example when the concentration of objects in the view is particularly biased to one corner or side. The leading diagonal scheme used for splitting up the viewport here is one of a number that can be arbitrarily implemented. In addition to the network and tilesort potential bottlenecks (which are both more likely to become problem points than in basic sort-first tiling), the compositing process now takes its place as a potentially very compute and data intensive task. Also, graphics acceleration hardware has not typically been optimized for the *glReadPixels* commands and can be a serious restriction on performance. The latest and future generation of accelerators based on the new PCI-express platform look set to improve this situation substantially, however.

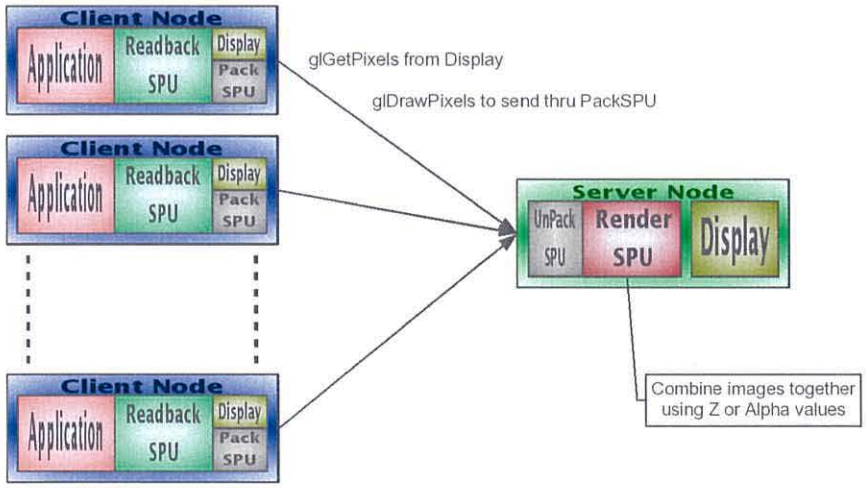


Figure 2.5: Chromium Sort-Last DAG Configuration

Figure 2.5 shows a sort-last configuration. Here, multiple applications (sources of visualization) are all rendered on a number of client nodes. The pixels resulting from this are sent over the network to a compositing server. This compositing server then puts the layers of graphics together in the appropriate order to produce the correct output. To establish the correct order, either Z or alpha values are sent from the client nodes alongside the pixel data. For each pixel in the output view, the compositing server can then blend or layer the

input appropriately to produce the correct output.

Synchronization and state are handled by Chromium in two ways [16]. Firstly, Chromium has a state tracker that can efficiently compare two graphics contexts and generate SPU calls for every difference. Secondly, Chromium adds two commands to the GL syntax for the purpose of synchronization over the network [17]. Server nodes use *crBarrierCreate* and *crBarrierExec* to keep the distributed components synchronized. Such server nodes receive command streams from multiple clients through which they will swap in a round robin style, waiting on a single stream until a barrier is received. Once all streams are handled, that particular barrier is satisfied and execution can proceed, probably with the next frame as the barrier synchronization would typically be called in the renderer prior to a *crSwapBuffers*.

Chromium additionally has a number of useful features. Firstly, a point-to-point network abstraction can be utilised by SPUs and custom applications alike in order to communicate with each other via non-standard routes, for the purposes of creating advanced compositing hierarchies. Secondly, the inheritance based nature of SPUs enables stylized drawing to be achieved through the creation of a processing unit that modifies the necessary GL commands to provide the required look.

Other Research

Staadt *et al.* present a review of several cluster rendering systems in [18], including data specifying the immense networking requirements for such work. Their study of OpenSG and VRJuggler shows absolute frame rates better than those of Chromium, but with associated longer startup times and modifications to the visualization codes required.

Samantra *et al.* [19] presents a hybrid sort-first/sort-last system implemented within the Windows operating system and a detailed study of the performance of this system in viewing three different geometry models. The dynamic, view-dependent partitioning improves upon static-partitioning, but at the expense of using a custom application that can only support static models due to the distribution of the model through the pipeline.

AnyGL [20] is a cluster visualization system built specifically for large-scale graphics applications involving huge scenes. It also implements a hybrid rendering system and a new state tracker architecture that aids the scalability of the pipeline, particularly between the geometry distribution and rendering stages of the pipeline.

Winkelholz and Alexander [21] present a virtual environment system that handles media beyond graphics including a Flock of Birds sensor-set and speech recognition. However, it takes advantage of the specialised data-flow graphs approach of implementing virtual environments in order to reduce the rendering load, hence restricting the generality of the application. It is worth noting that a combination of high-performance compute and commodity workstations is used in this system.

Lever *et al.* [22] discusses work carried out between the Manchester Visualization Centre and Advanced Visual Systems, Inc. on the adaption of the AVS/Express visualization toolkit to multiple graphics pipe equipped machines. Although this is not cluster parallelism, the management of delays between simulation code and rendering for real-time immersive applications is relevant. The use of an intermediate process between the application and rendering enables the parallelism of such high-end machines to be exploited and protects against interruption to the virtual experience when the application has not completed a frame by the time it needs to be rendered. In this latter case, AVS/Express Multi-Pipe Edition will re-render the previous frame but from the new viewpoint.

Commercial Products

The commercial market provides a number of offerings that are very relevant to this discussion of cluster rendering. However, being commercial, it is more difficult to establish firm details on some aspects of such systems.

ModViz, Inc. [2] is a company spun out from Siemens in order to provide supercomputer class visualization using computer clusters. Their core product, known as Virtual Graphics Platform [23], is an OpenGL standard based cluster visualization system that, similarly to Chromium, can operate either transparently to an application or with an application that can interact with the VGP system. The VGP software takes the OpenGL command stream and divides it up amongst the available render and compositing nodes as necessary to provide the display systems available. The render, composite or display components are dynamically instantiated on cluster nodes as required depending on the job at hand.

Sun Microsystems offers the Sun Fire Visual Grid product. This is a high-end hardware and software system that makes the power of several of Sun's top-end XVR-4000 graphics accelerators available as if it were a single system [24]. The software is less advanced than Chromium and is unremarkable. The hardware platform, however, is well-suited to scientific

[2] ModViz can be found on the web at `http://www.modviz.com`

visualization, but uses high-cost equipment. A master host runs the application and uses a Myrinet high-speed network to dispatch the graphics around amongst up to 32 graphics accelerators in 16 Sun Fire 880Z visualization servers. Each graphics accelerator can then drive up to two displays, for a maximum of up to 64 displays overall. The Myrinet network is dedicated to the 3D graphics data and a secondary Gigabit Ethernet is used for all other data. As the master node can be anything up to a 24 processor machine, the application using the Visual Grid can be large and complex.

IBM have recently announced their Deep Computing Visualization (DCV) effort. DCV encompasses an infrastructure for providing visualization in a scalable and manageable manner [25]. In practical terms, there are two setups that DCV initially supports. One is utilising a cluster to provide rendering. The other is in enabling remote visualization to a number of clients (collaborative workers). Again, DCV offers OpenGL support alongside a customization capability and is very similar to Chromium, as can be seen in Figure 2.6.

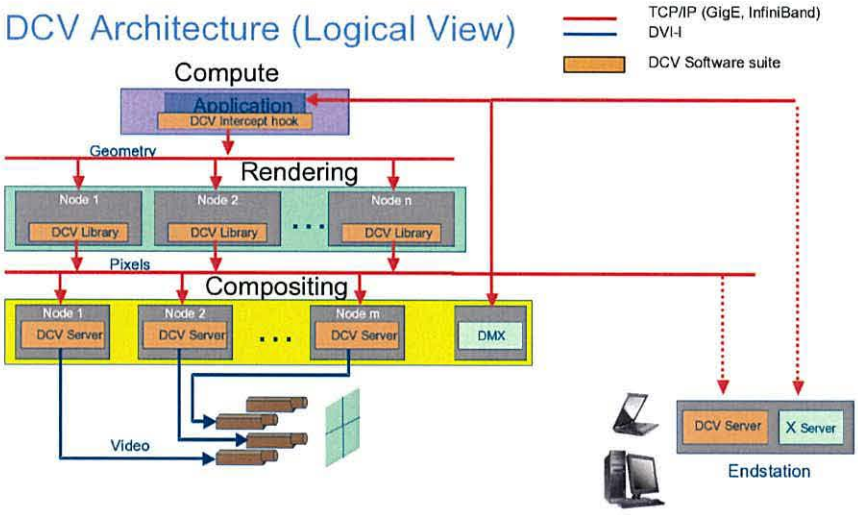


Figure 2.6: DCV Logical Configuration (Courtesy IBM)

IBM distinguish DCV from Chromium through several unique features, of which the non-marketing ones are:

- Support for some additional OpenGL extensions.
- Use of pixel shader blurring for compression.
- Enhanced security, including at the pixel level.
- Improved instrumentation and a performance dashboard.

Other than this, DCV once again offers nothing more than a pre-built and integrated setup of remarkable similarity to Chromium.

HP's solution will provide the Sepia high-performance compositing card as well as commodity hardware. Their end-product, known as the Scalable High-End Visualization System (SHV), can scale from eight to thousands of nodes through the use of the Sepia hardware and a dedicated compositing network, instead of software, for gathering the results of the parallel rendering [26]. Sepia works in much the same way as the old first-generation 3DFX graphics accelerators did. To overcome the performance limitation of reading back pixel data from the frame buffer, the DVI video feed goes from the graphics accelerator on a particular machine to the Sepia card on the same machine, which then outputs directly to the dedicated Infiniband compositing network. This also means that the PCI bus of the machine is not saturated in any way and allows the full rendering power of the machine to be utilized.

HP's combination of commodity and custom hardware presents a very high-performance and adaptable solution in which the cost overhead is limited.

A disadvantage of all commercial offerings is the cost of such systems compared to a custom-built Chromium based solution. However, the availability of different pre-built systems will enable the field to take off, prices to drop and companies and institutions alike to simply buy-in a complete solution without needing high-end expertise in the area. Of course, even in such a situation as having expertise available to install and setup such a system, there is the cost of the manpower involved. Balancing of this cost with the overhead of a bought-in, pre-built solution will be increasingly even in years to come. As the extraordinary pace in the development of graphics hardware begins to slow, these solutions will become necessary for ever greater performance requirements.

2.2 The Grid

As much as the term "the Web" came to represent the second generation of the Internet-enabled world, "The Grid" encompasses a whole host of technologies and ideas that will be the third. Many of these next-generation ideals may look and handle similarly to the present-day, but the Grid represents the necessary new way of thinking about what goes on behind the scenes. The effect will be to make computers and the networks on which they reside usable in a different, more integrated way than is presently the case. This will

truly represent the "electronic underpinning for a global society in business, government, research, science and entertainment" [1].

The ultimate future of Grid technologies is to produce large-scale dynamic systems that couple geographically and architecturally distributed facilities to provide access to computation, storage, instruments and other devices. In order to enable easy-to-use access, a complex system of software layers is necessary that enables integration with established real-life policies and practices. Beyond this, the Grid represents an abstraction of the physical facilities into a uniformly accessible virtual system where the limitations are no-longer imposed by the technology, but by the intricacies of society.

2.2.1 Origins of the Grid

The 1980s was a decade of research on parallel systems. Through the development and deployment of hardware, software and algorithms, it soon became apparent that such research would push the bounds of what was possible on the individual computers of the time. This resulted in work looking at breaking free of the boundaries of single machines and splitting tasks across distributed systems. The early 1990s followed up with work on parallel APIs such as PVM, MPI and OpenMP. This work harnessed both shared and distributed memory systems. The types of parallelism exhibited by certain problems and algorithmic solutions led to the creation of the first cheap computer clusters, known as "beowulfs". When the idea of a Grid first came up at this time, it was envisaged as a way to extend parallelism over more distributed systems in order to create ever large virtual supercomputers.

Alongside all this work in parallelism, the traditional Grand Challenge key problems in science began to be tackled by researchers from multiple disciplines and across geographical and institutional boundaries. Interdisciplinary research also blossomed and formed a number of distinct, new research areas such as bioinformatics [1]. These key ideas led to new ways of working for large problems. New ideas came forth to tackle the security, accessibility, coordination and distribution problems created. These are all things that would become part of the Grid philosophy.

At SuperComputing 95, the first showing occurred of what came to be known as Grid ideas. The Information Wide-Area Year (I-WAY) [27] project aggregated more than 17 sites in the US together via the vBNS (very high speed Backbone Network Services) [28] and one of many pieces of software deployed on I-WAY was an early Grid system providing capabilities in the areas of accessibility control, security enforcement, resource collaboration and data

distribution. This focus on the very dissimilar Grid concepts was an important stepping stone from the throws of distributed computing research at the time. The introduction of new problems and focuses began a new era in research for providing distributed computing in an integrated and managed way. Grid computing is very much distributed computing for today's highly-connected world.

I-WAY led on to the creation of much Grid research. Many projects popped up investigating the full range of services and components of a Grid. The Globus and Legion projects tackled the problem of creating an entire Grid infrastructure meeting the requirements initially considered through and from I-WAY. The high-performance aspect was not lost, however, as the Condor project tackled high-throughput, distributed scheduling and several other projects tackled high-performance scheduling (CITES for Apples, APST, MArs, Prophet). Additionally, the Network Weather Service project began to study resource monitoring and prediction, the Storage Resource Broker (SRB) began to tackle the problems of data access and the NetSolve project looked at client-server remote computation.

In the late 1990s, Grid researchers from around the world came together to form the Global Grid Forum (GGF) [3]. The GGF provides the necessary system for the interchange of ideas and discussion among Grid researchers in the effort to produce the standards base for Grids. The GGF also states the following as a high-level description of Grid computing:

> Grid computing is increasingly being viewed as the next phase of distributed computing. Built on pervasive Internet standards, Grid computing enables organizations to share computing and information resources across department and organizational boundaries in a secure, highly efficient manner.
>
> Organizations around the world are utilizing Grid computing today in such diverse areas as collaborative scientific research, drug discovery, financial risk analysis, and product design. Grid computing enables research-oriented organizations to solve problems that were infeasible to solve due to computing and data-integration constraints. Grids also reduce costs through automation and improved IT resource utilization. Finally, Grid computing can increase an organization's agility enabling more efficient business processes and greater responsiveness to change. Over time Grid computing will enable a more flexible, efficient and utility-like global computing infrastructure.
>
> The key to realizing the benefits of Grid computing is standardization, so that the diverse resources that make up a modern computing environment can be

[3] http://www.gridforum.org

discovered, accessed, allocated, monitored, and in general managed as a single virtual system even when provided by different vendors and/or operated by different organizations.

Requirements of the Grid

There are a number of different views on what services a Grid software layer, or middleware, should provide. However, there is increasingly broad agreement over the necessary core set. Indeed, parts of this have been standardised through the GGF, as will be covered later.

Security and Authentication The cross-administrative domain and multi-layer nature of Grids presents a difficult problem in authentication and authorization. No longer are we concerned with a situation where one client authenticates to access the resources of one server. Instead, we have multiple tiers of systems that act as clients and servers for each other and within which any individual user may or may not have rights of access. Take an example where a user on one machine tries to launch a job on a second machine to process data held somewhere else. Firstly, machines one and two must establish some sort of trust that they are some form of collection. Secondly, the user authenticates as the client to the server on the second machine in order to launch the job. However, the second machine then must authenticate as a client with the third machine acting as the server to access the data to process. Whilst simple when involving just three machines, the task becomes much more complex when scaled up. The particular requirements [29] are the ability to Single Sign-On (it is absurd to expect the user to re-enter a password or such like for each authentication), the Mapping between such an authentication scheme and local operating system authentication (crossing administrative domains where the operating system has different users), Delegation (so that the many layers are able to authenticate and authorize as necessary, alongside the risks of propagating such information) and Policy (only certain people and/or groups being able to access certain things).

Scheduling and Resource Allocation Managing the allocation of resources such as processors, memory and storage between the jobs that need to utilize them in an efficient way is both an important and difficult task. To the user, the scheduling process should be transparent and they should interact with it only to submit jobs [30]. The actual process would need to involve no single scheduler, but a network of them, all interacting with local operating system schedulers to allocate resources, create processes and deallocate resources.

Data Transfer and Communication In a system as meshed together as the Grid, there are three requirements of file systems [31]. (i) Users require access to status information on many different sites. (ii) Users require access to data and executable files on many different sites. (iii) Job executables must be able to read and write very large data sets. Thus, we require consistency, reliability, security and performance that places unique requirements on both the localised fabric (disks and their access patterns) and the interconnections between many distributed sites.

Resource Monitoring and Discovery Once again, the scale of production Grids introduces new intricacies to established areas of distributed computing. Establishing a reliable architecture in which to carry out resource information sharing is a difficult task. The tasks involved include exchange of soft-state data, searching of information directories and monitoring for failure and of performance.

Heterogeneity The Grid concept is all about virtualization. The user should not know that there are x machines being contacted for their job - they should just submit the job and get back the results from the illusionary, single, extremely powerful, computer. However, the world consists of multiple operating systems and processor architectures, many different network structures and speeds, and a great variety of people with different views and policies. If the behavior of the Grid from the user's perspective were to change due to any of these factors, then the Grid has failed to act as the single-being, omnipotent/omnipresent system that is envisaged.

2.2.2 Grid Generations and Middleware Development

The Community Grid Model

Berman, Fox and Hey present the Community Grid Model in [1]. Grid researchers from all walks of life have moved towards this layered model in order to enable easier development and integration of the complex systems that comprise Grid resources.

As shown in Figure 2.7, the four horizontal layers here represent a traditional, technical way of looking at the Grid, akin to other layered approaches. It is fairly easy to understand these layers and their principal components. However, the two vertical layers represent relative unknowns that will impact upon the horizontal layers and the Grid that they represent in the future.

The lowest horizontal layer represents the hardware resources that form the Grid. Thus,

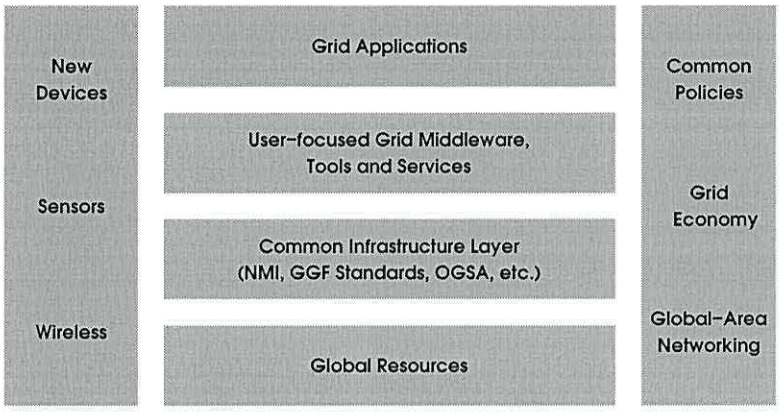


Figure 2.7: Layers of the Community Grid Model (from [1])

networks, storage, compute and visualization devices all fit into this layer along with other devices. By definition, services (a Grid service is a facility or resource accessed via Grid protocols) will come and go in a very dynamic way in the Grid and the machines themselves will vary enormously in location and performance capability. The two middle layers represent the software that sits between the hardware and the user interaction with the Grid. The lower middle layer represents the software services and standards that are followed on all Grid members - such as OGSA and WSRF, as will be discussed below. These community agreed standards provide for the interaction amongst nodes in order to represent the Grid as a virtual machine and share the resources of it. The upper middle layer represents an abstraction away from the details of the lower middle layer in order to provide services to applications. This easing of the difficulty of data transfer, remote access, scheduling, etc. provides a much more convenient environment for actual application development and deployment. The user parts of middleware sit at this layer whereas the lower-level, more technical parts exist in the lower middle layer. The top horizontal layer represents the applications that use Grid services and the users thereof. Although all the building blocks of the Grid exist at lower layers, if this layer does not work then the Grid is useless. The dependence on the user community establishing how to make use of the services available to form a Grid application (whether it uses one Grid service or one hundred) cannot be underestimated and there is much work to do in this area.

The proliferation of network devices alongside the adoption of key networking technologies such as IPv6 will affect all the horizontal layers. As such new devices and sensors are attached to networks that themselves become increasingly dynamic through the provision of wireless, cellular components, the Grid software will need to adapt to the changing world. At the same time as the Grid itself becomes more and more ubiquitous, there will be

a large requirement for further development in the high-level management facilities available regarding the impact of a global network and its implications for the global economy and sharing policy. The socio-economic impact needs to be carefully managed and developed as we head towards a single global Grid that assimilates such different nations and societies, even if we don't get all the way there.

Evolution

The previously mentioned I-WAY project led directly into the Globus project. However, it should be mentioned that I-WAY itself grew out of other projects. The earliest recognisable efforts were projects to link together the US supercomputing sites in the early to mid 1990s. It was at this time that the term 'metacomputing' came about in association with the US Gigabit testbed networks and their use in making the high-performance compute facilities available to a wider audience. It was around this time that the I-WAY project, and another project called FAFNER [32], began. FAFNER was aimed at factoring RSA130 encryption and was accessed through a CGI web interface. This interface provided a range of new functionality, including cluster management and live status reports. Additionally, FAFNER utilised a new factoring method called Number Field Sieve (NFS) that enabled even desktop computers to carry out useful computation, contrary to I-WAY's use of large supercomputers and their related resources. The well-known SETI@Home project [33] is an example of the FAFNER principles taken forward.

The key enabling technologies in moving on to producing what we now call a Grid infrastructure have been the bandwidth explosion in networking technologies and the cooperation of separate research groups to produce standards. This has enabled the Grid concepts to become truly realistic. Foster and Kesselman are rightly thought of as 'the great makers' and their seminal collection [34] from 1998 is justifiably thought of as a key moment. In order to provide a heterogeneous, scalable and adaptable Grid, the underlying infrastructure (i.e. within the middle two horizontal layers of the Community Grid Model) needs to provide a number of main data and computation features, as follows [30]:

Administrative Hierarchy This determines how administrative information distributes throughout the Grid and how far-fetching its effects are.

Communication Services The requirements of networks in the Grid era include the full range of modern networking technologies. Point-to-point, multicast and broadcast

links, both in reliable and unreliable forms alongside various Quality of Service parameters are needed.

Information Services Such is the dynamism of services in a Grid that we need information exchange services to provide sharing of service status, location and resource requirements.

Naming Services Whilst related to Information Services, Naming Services are different in that they provide a uniform namespace over the entire Grid environment.

Distributed File Systems It is key that Grid applications have access to data among many servers. In order to enable such an environment, we need a uniform file system enabling data access in potentially varied and optimized ways to any user-machine combination in the Grid that has the necessary security credentials.

Security A Grid environment requires all aspects of security to be tackled. Data confidentiality, authentication of users and machines, authorization of users and machines, and accounting are all critical.

Monitoring Systems To enable high performance and reliability, it is necessary to monitor and report upon Grid services at layers from middleware up to application.

Resource Scheduling To handle the resources available in a Grid in an efficient, fair and policy-controlled manner, it is necessary to have advanced, yet transparent, scheduling between the user and resources. The layers of such scheduling systems from operating system all the way up to Grid application must interact sensibly.

As these formal Grid efforts intensified in the late 90s, the second-generation of the Grid came about, initially in the form of the Globus project. The first Grid standards reference implementation produced was built on the standard Unix services of SSH (the Secure SHell), LDAP (the Lightweight Data Access Protocol) and FTP (the File Transfer Protocol) with some enhancements, principally to do with authentication. Known as the Globus Toolkit, the software implements the basic Grid services in a modular way to provide a single virtual machine. Through the provision of these basic services and well-defined Application Programmer Interfaces (APIs), a wide variety of applications can be developed based on different methodologies and not any single one such as the Object-Oriented (OO) model. Version 1 of the Globus Toolkit evolved from the I-WAY project and contained the basic set of services. Version 2 became more of a standard and continues to evolve and be used today as a result of the level of takeup. The work on this thesis is implemented with GT2, as this has been and continues to be the basis of the UK e-Science Grid.

Within the same time frame as the Globus project was becoming successful in the United States, the UNiform Interface to COmputer REsources (UNICORE) project [35] was created in Germany. This has become more established in Europe and has been key to the creation of the EUROGRID and GRIP projects. UNICORE is unlike the earlier versions of Globus in that it is a much more integrated system. From an external viewpoint, UNICORE considers all the necessary Grid components but it also utilises existing web technology and the Java language to provide a uniform, familiar graphical user interface. At a lower level, application jobs within a UNICORE Grid take the form of recursive objects containing groups of tasks and also contain additional information specifying dependencies and destination systems. This dual high and low level structure is attained through the encapsulation and serialization abilities that Java provides in order to allow the authentication and initial access at the byte-layer and the job transfer at the object-layer.

Also around the same time period, there were a number of significant other pieces of work taking place. Legion [36], from the University of Virginia, is an object-based system for accessing distributed, high-performance computers. Legion presents the user with a single virtual infrastructure, irrespective of the actual underlying hardware and differentiated from Globus by being object-based. Legion has been commercialised as part of the Avaki company. A number of job submission systems were also around at this time. Five popular ones are:

Condor [37], which has been in development for many years and is a tool for harnessing the power of idle workstations and scheduling submitted job across them. Condor can also handle checkpointing and process migration as jobs must be linked against the Condor libraries themselves.

The Load Sharing Facility (LSF) [4] is a commercial system used for distributed batch submission and load balancing in a heterogeneous environment.

The Portable Batch System (PBS) [5] is another batch queueing system that provides particularly good control over administrative policy including time policy and access policy.

Sun Grid Engine (SGE), [6] formerly developed as Codine, is another very similar product from the user's perspective. SGE allows the user to specify the requirements for

[4] `http://www.platform.com/Products/Platform.LSF.Family/`

[5] `http://www.openpbs.org`

[6] `http://www.sun.com/software/gridware/`

the job they wish to submit in a fairly complex way and then allocates that job appropriately.

Load Leveler is a parallel job scheduling system produced by IBM for AIX and Linux. It is aimed at dynamically scheduling, optimizing utilization and providing control of jobs submitted to a *Load Leveler cluster* - a group of machines running the Load Leveler daemons.

In summary, the second generation of Grid software made the key moves in bringing Grid development into the mainstream. Evolving from the original versions of the Globus Toolkit through to GT2.4 provided the integration required for moving from compute intensive jobs through to more open deployments. As part of this, various software was created to provide higher-level services.

Third generation Grid technology has moved on. In order to provide the desired component architecture and information resources, it was necessary to adopt a more service-oriented model and pay more attention to metadata [30]. This goes on to have significant implications for the underlying technologies. Such changes re-focused middleware development away from solely thinking about massive compute resources towards the users and the machines on their desks. This, in turn, has further effects. The increased likelihood of failure led to more work in recovery and failover. A larger number of machines requires more work on coordinating the resources available. This has all come together to form a more automated Grid that no longer requires human interaction as humans would not longer be able to cope with the scale of things now involved.

In 2002, the Globus Consortium worked alongside IBM to create this third generation Grid. The result of this relationship was the Open Grid Services Architecture that combines web services and Grid technology. OGSA now defines core services as such [1]:

- Systems Management and Automation
- Workload/Performance Management
- Security
- Availability/Service Management
- Logical Resource Management
- Clustering Services

- Connectivity Management
- Physical Resource Management

This change of emphasis morphs middleware into a service framework in which all resources are virtualized as Grid services and accessed through the Web Services Definition Language (WSDL) [38]. Instead of the focus being on interfacing with resource controllers, the increased user-centric nature of Grid third generation shifts to focus on the hosting environments (often portal like) for accessing Grid services [39]. This in turn radically changes the environment in which Grid developers operate - now, the principal tools are web based, in particular Apache AXIS and Microsoft .NET. The Globus implementation of OGSA, which included AXIS, was called Globus Toolkit 3 (GT3) and intended to preserve functionality from GT2 as far as possible. 2003 saw the GGF produce the first Open Grid Services Infrastructure (OGSI) standard, which moved beyond those relevant standards laid down previously in order to further integrate the web services approach into the Grid.

However, things didn't stabilise there. Microsoft joined the team of Globus and IBM to produce Grid generation 4 - the Web Services Reference Framework (WSRF) [40] - in order to better deal with the requirements of business, consumers and scientists together. The most significant differences between WSRF and OGSA are that WSRF allows access to resources (as GT2 did) through a web services infrastructure and that WSRF services do not have state whereas OGSA services did. Continuing tradition, the Globus Toolkit version 4 (GT4) [41] reached alpha status in late 2004 and final release on the 29th April 2005. The current stable release is 4.0.1. WSRF is arguably less revolutionary and more evolutionary, but sensibly so. All the work that business and academia has put into the web in the past years is not without value in a Grid environment and should accelerate take-up of the Grid, even if people do not actively realise what they are getting into. GT2 still remains very popular precisely because of the advantage it has for science and the volatility in what comprised the next generation Grid standard. Perhaps GT4 can finally change that in the years to come.

Underneath the Globus 2 Hood

Whilst it is important to remember that the three significant generations of the Globus Toolkit (2, 3 and 4) have all been seeded by different Grid standards, it is also important to note that the component based architecture implies some commonality. We cover here the components that comprise our UK e-Science based Globus Toolkit 2 working environment.

GSI Globus utilises the Grid Security Initiative (GSI) to provide security at the various interfacing stages that occur [7]. GSI is an implementation of the Generic Security Service (GSS) described in [42] that itself provides an application programmer interface to access its own standard procedures for credential exchange, authentication and encryption. GSS sits on top of established systems such as SSL in order to provide these functions in a standardized manner and Globus implements GSS so that different underlying security mechanisms can be used simultaneously. Globus also provides the ability to query what security mechanisms are available in order to select one to use. GSI therefore is one of the Globus core components.

Grid Resource Allocation The Globus Resource Allocation Manager (GRAM) is the core, low-level service that is responsible for resource allocation and control. The fine detail of making resources available according to some policy is typically handled by an underlying resource management system, such as Condor or LSF. Submitting and executing any job that does not use one of the compatible/integrated job managers or requires its own execution environment (e.g. Java, Perl, Python) is very difficult and often restricted by this lack of integration. A GRAM provides access to a group of resources controlled by the same policy, for example a cluster as a whole managed by Condor. GRAMs represent the standardized interface to accessing resources in a Grid. The Resource Specification Language (RSL) is used to express requirements for a job (for example, x free processors) and the RSL produced by a Grid application may involve the necessary use of several GRAMs. The RSL is therefore handled by a resource broker that is responsible for splitting it up and extracting the specifics of GRAM submission(s) from the high-level RSL syntax. Multiple brokers may be involved in a single request through the use of application-specific brokers to translate an application's requirements into specific individual resources and then resource brokers to locate suitable resources. These specifications are then passed to *co-allocators* that are responsible for passing all components of the request onto individual *resource managers*. These are the components that talk to local job managers and recognise two types of parameters:

- MDS attribute names express resource requirements.
- Scheduler attributes express job requirements.

Hence, a multi-resource RSL request, as described in [2], may look something like:

```
+((&(count=5)(memory>=256M)
   (executable=ls)
```

[7] For example, authentication when connecting to a remote resource.

```
  (resourcemanager=escher.sees.bangor.ac.uk:4000))
(&(count=2)(cpus>1)
  (executable=w)
  (resourcemanager=ainur.bangor.ac.uk:4000)))
```

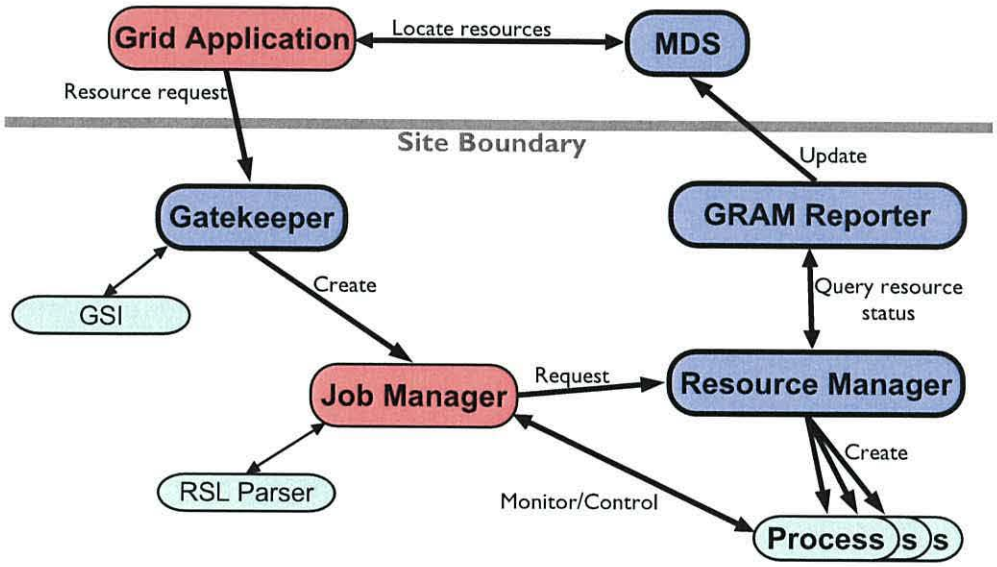


Figure 2.8: GRAM Implementation (Adapted from [2])

Figure 2.8 shows GRAM in implementation. GRAM may either be extensively used to provide service to a Grid application or such an application may choose to do more work for itself. When utilising GRAM, it provides a client API library that can be used by an application to authenticate with a remote site and transfer a job request [2]. This takes place via a *gatekeeper* that acts at a site to authenticate requests and launch job managers as the local users. In the case of the UK e-Science approach, GSI public keys are mapped to Unix users on a site-by-site basis. This is a time-consuming, inefficient and non-scalable approach that is the subject of an entire branch of research. Once the job manager is launched at the remote site, it is responsible for creating the necessary processes to fulfil the request; be it through submission to something like Condor or a simple *fork*. It then monitors running jobs and notifies the application of state and termination. Once the job finishes, the job manager also completes. The *GRAM reporter* shown on the diagram is responsible for updating the information in the Grid information system and always runs, even when no jobs are.

GridFTP GridFTP [43] is an extension of the FTP protocol to provide a high-quality data

transfer mechanism within Data Grids, which are Grids with applications based around large amounts of data being accessed intensively. A typical example application is a large-scale, distributed physics application in which sizable facilities (such as experiments producing data and supercomputers processing it) need to share data. GridFTP aims to provide a reliable and high-performance system for such uses and also provide support for replication of data amongst a server hierarchy. GridFTP integrates with the Grid Security Infrastructure of Globus and can also provide striped data transfer (i.e. sourcing from multiple servers) for improved performance [8]. The replication capabilities consists of several software layers to provide management, consistency and rollback. A GridFTP server is a standard component of a Globus Grid installation.

Grid Data Storage It is an implicit part of Grid computing that jobs will execute on machines at a widely dispersed set of sites. It is therefore necessary to move the data for such jobs around among these sites fast enough to not limit high-performance jobs. The Global Access to Secondary Storage (GASS) [44]system aims to provide this by satisfying the following defined Grid I/O requirements:

Uniform access to files - overcoming the complexities of authentication, communication protocols, naming, etc to provide the same access mechanisms to data, irrespective of location.

Diverse data sources - providing access to a variety of data sources including HTTP, FTP, etc.

Dynamic resource set - allowing for the potentially very dynamic set of users and services.

Support for streaming I/O - providing Unix command-line like streams access in order to allow existing applications to be more easily transferred to a Grid environment and chaining together of the components thereof.

Little or no program modification - to reduce the cost of deploying Grid versions of existing software.

Support for programmer-directed performance optimization - allowing application programmers to override efficient default I/O strategies in order to optimize performance for their specific application.

Existing distributed file systems are often time-consuming and complex to deploy. For example, the Andrew File System (AFS) [9] is a kernel-level service (leading to relatively

[8] 500MBit/sec reported in the year 2000!

[9] `http://www.openafs.org`

high performance) in which matching file access permissions in a multi-institution environment would be exceptionally difficult. The Prospero File System provides a focus on management and organization of files, but not the level of performance required. GASS does not aim to support all types of data access, but instead focuses on I/O patterns that are used in high-performance computing. These four patterns are:

- Read-only access to an entire file, e.g. for multiple database users.
- Append-only write access to a file, e.g. for monitoring via log files.
- Non-consistent shared write access to a file, e.g. in which any of the writes from a multiple of writers is valid.
- Unrestricted read/write access to a file, e.g. in which only one process is accessing the file at a time.

GASS utilizes file locks and local secondary cacheing alongside pre-staging and post-staging in order to handle access to both large and small files in a high-performance manner. Some file accesses will simply be streams over a network and so will not be cached. However, GASS allows the user to control some of the details of cacheing when it does occur, including the location where a file may be cached, which can have performance advantages. From the Grid application perspective, GASS controlled file access is performed using standard Unix methods. At its most basic, read and write calls need not be modified, but open and close calls will have URLs passed to them rather than file specifiers. However, GASS does provide more complex lower-level APIs for those Grid applications which choose to use it, in order to optimize cache management, client access details (e.g. proxies, duplication, Maximum Transmission Unit sizes) and server details. The Globus Toolkit 2's native command line tools utilize GASS for basic functionality. The base *globusrun* command uses GASS to handle standard output and error streams and to stage scripts and executables. This stream management has the advantage of making use of the GASS cache in order to provide buffering. Tests have shown [44] that GASS's file locking leads to slightly lower performance than AFS for small files but also that GASS performs better in dealing with larger data files, as would typically be the case with a high-performance job.

Grid Information Services The Grid allows geographically dispersed access to and use of resources. Some of these resources are well-known and permanent, others are highly dynamic and very temporary. Enabling users and potential Grid applications to have knowledge of the resources that are made available within their Virtual Organization

is a non-trivial task. This is why Grid Information Services (GIS) exist - to provide discovery of, information about and monitoring of available Grid resources. These are vital to provide a dynamic Grid that supports resources appearing and disappearing, scheduling amongst such resources and observation of the performance of resources being used. GIS provides a Lightweight Data Access Protocol (LDAP) based architecture for tackling these problems composed of two basic elements:

Information Providers, that provide information about resources in an LDAP structured model of attribute-value pairs.

Directory Services, that collect and provide information gained from information providers. Specifically, Aggregate Directory Servers that collect virtual organization scale information and provide views and searching capabilities through LDAP interfaces.

The Globus Toolkit 2's standard GIS component is the MetaDirectory Services 2 (MDS-2). MDS-2 consists of an information provider component called a Grid Resource Information Service (GRIS) and an aggregate directory service component called a Grid Index Information System (GIIS). In order to provide a timely system, MDS-2 requires that information providers are light and provide suitable time-related information alongside their resource-related data. To ensure a robust system, MDS-2 specifies that the failure of any component should not restrict information sharing on other parts of the system. This requires that information providers are highly distributed and preferably located in the same place as the services they report on and that all components assume failure to be the rule rather than the exception. Ensuring that an MDS-2 system is implementable in reality necessitates that policy controls can be put into place in order to restrict certain privileged information and also to contribute to the scalability of such a system. The multi-tiered virtual organization hierarchy simplifies this.

MDS-2 contains two protocols that form the basis on which all information is exchanged [45]. Firstly, the GRid Information Protocol (GRIP) is used to access information on actual resources from a GRIS, and secondly the GRid Registration Protocol (GRRP) that is used to register such a provider with a GIIS server. The GIIS uses GRIP and GRRP to obtain information on a resource from a GRIS, store such information and provide a query interface for searching. Because of their aggregating nature, GIISs provide both scalability and scope to information within virtual organizations. The use of LDAP as the GRIP protocol gives several advantages, as discussed below.

Object classes represent names for different types of resources (filters) and *distinguished names* provide the structure and hierarchy that groups object classes together

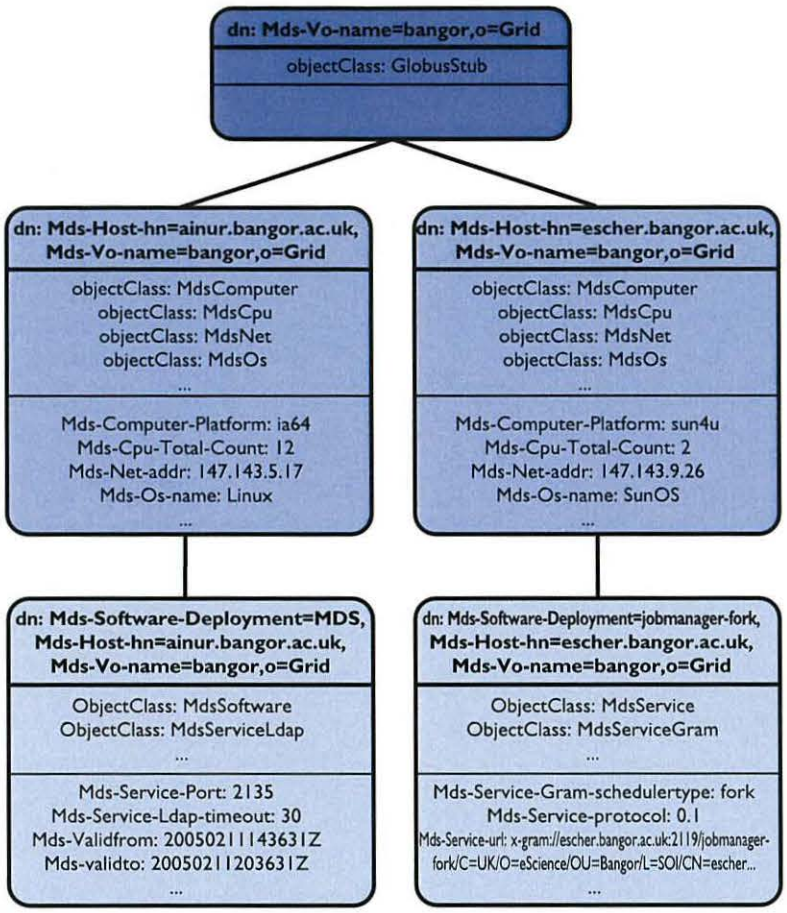


Figure 2.9: GRIP protocol data in the LDAP hierarchical namespace.

in a queryable way. Such searches can then be filtered and only selected attribute subsets retrieved. Figure 2.9 shows the hierarchy and application of object classes to different components of that hierarchy. The GRRP protocol dictates that messages contain validity timestamps so that the information will eventually be dropped by a directory unless refreshed, effectively determining that a failure has occurred without having to be notified explicitly as such. Through the multi-layered application of the GRRP protocol, MDS-2 produces a hierarchy of aggregate directory servers. In the same way that a GRIS registers its information with a GIIS, that GIIS can then register its information with a higher up GIIS. Such higher up directories will not maintain all the data, but will know where it can be obtained from and be able to retrieve it should they be asked for it. This has the advantages of lowering storage requirement and machine load for a higher-level directory as well as allowing specialized policy to be implemented on them, for example querying known trouble-spots earlier

than those that have been more reliable. Security in an MDS-2 setup is provided by the standard GSI, in order to have public-key authenticated communication between clients and servers. By utilising OpenLDAP for MDS-2, such security is easily implemented through the plugable Simple Authentication and Security Layer (SASL) and its provision of the GSS-API. GRISs are implemented as OpenLDAP backends that call information providers and then merge and maintain that information, which is typically for the host on which it is running. The information providers are called via either Unix shell scripts or through loadable modules that run within the server itself and not in a subshell. The loadable modules option is more efficient but also more difficult to implement. Information gained will be held for an administrator-configured time before being refreshed. GIISs are also OpenLDAP backends. However, the administrator sets these up to collect together information from information providers and other GIISs and merge this together into an LDAP tree that is then navigated using distinguished names.

Commodity Grid Toolkits The areas of commodity computing and high-performance computing have evolved in parallel for many years. However, with the importance of the Grid clearly likely to increase in the future, more of a commodity nature needs to be brought in to make it more accessible. However, these two fields have evolved with very different targets. The Commodity Grid project was created to combine these two areas together to the advantage of both. The use of commodity components and ideas within Grids will ease and enable development of Grid applications. The availability of what comprises the Grid will enhance commodity computing by definition. Whilst the Commodity Grid project intended to produce toolkits for various commodity platforms, including Perl, Python, CORBA and DCOM [46], at this time there are toolkits for Java, Python, Matlab and Portal development. Java was an obvious first choice for toolkit development. Commodity computing has taken on new meaning with the advent of Java - where code can run on such a diverse selection of hardware platforms and operating systems. Indeed, the reasons for Java being a good language for Grid development are fairly obvious and provide comparison with C/C++ as follows:

The Java Class Library - is a large, deep resource of elements that provide functionality from SSL to GUIs. C/C++ also has an extensive library of largely compatible (subject, principally, to the evolution of the language standards and compilers) libraries that provide equally diverse abilities.

JavaBeans - provide a component based architecture for development and deployment.

Bytecode Deployment - to enable cross-platform development and deployment of code very easily. This compares with the machine architecture and operating system specific binaries of C/C++.

Performance - approaching that of C and Fortran, when used in an optimized manner and with the HotSpot capable compiler. The overhead of running a Java VM produces slowdown, however.

Popularity & Diversity - with Java appearing in hardware from PDAs and mobile phones to Java Cards (credit card sized chip cards with Java technology) and web pages. In particular, Java applets enable ease of creation of Grid portals.

The Commodity Grid Toolkits are mappings of Grid functionality into commodity computing frameworks and environments [47]. In the case of the Java CoG, there are four levels of components that each utilise the abilities of those below:

Low-level Grid Interface - components to allow direct Java language interfacing with MDS, GRAM, GASS and GridFTP through GSI.

Low-level Grid Utilities - components to carry out common functions through easy calling within Java. For example, searching an MDS to find resources matching a specific RSL.

Low-level GUI Components - graphical components for common functions. For example, LDAP browsers, RSL editors and GSI public-key passphrase entry systems.

Application GUI Components - graphical applications that show Grid functionality and may be used to interact with it. An example here is the Grid Desktop that allows the user to drag various components between LDAP windows, RSL submission windows and machine representations (e.g.: an icon) in order to launch the said jobs on those machines.

It is through this structure that the commodity grid toolkits enable the more rapid pickup of Grid technologies and deployment of Grid applications. They are certainly useful and crucial to Grid work and are being developed onward for the new generation of the Grid - itself a lot more commodity based as it is.

2.2.3 Grid Practicalities

Grid research has been primarily based in academic and scientific facilities. As such, cutting-edge compute and network facilities have been part of the research and the Grids that have

been implemented. As the field blossoms, increasing numbers of commercial interests naturally become involved and Grid research permeates more into every day use.

As High Performance Computing Facilitators The Top 500 Supercomputers list, now in its 25th revision, continues to represent the massive investment in scientific research computers around the world. Some of the supercomputers on this list are components of Grids today and will be so increasingly in the future. Whilst many of these are not currently available within a Grid (or even on the Internet), they give some idea of the level of facilities that will be available in the future. The current top three supercomputer are as follows:

The Earth Simulator - An NEC machine in Japan, the Earth Simulator appeared by surprise to take the lead in the Top 500 from the traditional holders of the title, the ASCI computers in the USA, in June 2002. Consisting of 5120 NEC vector processors, it achieves 35.86 Teraflops Linpack performance against a theoretical maximum of 40.96 Teraflops over a crossbar network that directly connects each of 640 processor nodes to every other via one of 128 switch units, involving over 83 thousand interconnecting cables.

Nasa Columbia - Named in tribute to the shuttle and crew lost in space shuttle mission STS-107, Columbia [48] is a cluster of 20 SGI Altix 512-processor systems with 20 Terabytes of total memory. It achieved a Linpack result of 51.87 Teraflops (vs 60 Teraflops theoretical maximum performance) and connects the 20 nodes together using Infiniband networking.

IBM BlueGene - Developed by IBM for the US Department of Energy at Laurence Livermore Laboratories, BlueGene makes use of massive parallelism of slower processors. The current BlueGene/L [49] uses 65 thousand processors to provide a Linpack result of 136.8 Teraflops against a theoretical maximum of 183.5 Teraflops. The final machine will have a peak performance of 360 Teraflops.

Over High-Performance Networks Grids are based on ideas of sharing and connectivity. The future connected world will therefore rely on ever higher performance networks and the crossover between industry and academia has been particular apparent in developing them. The balance of performance between desktop links and transcontinental links remains fairly similar although it seems fair to say that bandwidth has increased at a rate beyond the demand for it. Latencies, however, are unlikely to improve much due to the limits of present-day physics.

The UK, USA and Europe each have networks that exhibit the same 10 Gigabits/second performance in the forms of SuperJanet 4, Abilene and GEANT. See figures in Appendix C for topology diagrams of these networks. Between these networks, there are also large pipes and increasing capacity. The Global Terabit Research Network (GTRN) aims to provide Terabit performance between many of these regional area networks in 2006, but is just one example. It is telling, however, that the requirements documentation for the next version of the SuperJanet UK academic network [10] places more focus on adaptability and diversity of use than on absolute performance. This is indicative of the utilization that has been seen with regard to SuperJanet 4 and reflects the changing ways of using networks, with technologies such as multicast reducing the level of infrastructure necessary to support a certain number of users. Of course, as an ever-growing number of networks exist, massive global performance (see Figure C.7) comes to enable the future of the connected world. The present-day Global Lambda Integrated Facility [50], is an example of this that is closely associated with Grids and thus demonstrates the integration of the two fields.

Real Grids Real-world 'production' Grids have been created in a number of environments. Good examples are the UK National Grid Service (previously known as UK e-Science Grid) [11] and the Nasa Information Power Grid [12]. A Grid is the connection of geographically and organisationally distributed resources to facilitate large-scale scientific research. As detailed in [51], the construction of a Grid depends on its intended use, such is the component nature of Grid software. However, a Grid will have its own Virtual Organization and Certificate Authority, require consideration of data-locality and communication issues, and need a great deal of cooperation between the involved real organizations. These relatively early Grids provide valuable experience of deployment realities that is relevant to all Grid research and has been relevant to us as we set up our small part of the UK e-Science Grid. Many present-day Grids are application specialised. The general-purpose Grids, such as the UK NGS and TeraGrid in the USA, are not financially solvent [52] and, as a result, a 'market economy' for trading of facilities in and amongst Grid operators is a solution. By enabling application-focused Grids to share and interact in this manner, the capabilities offered to users of such a federated Grid are extended.

Grid Futures The field of Grid research continues to develop and diversify. As well as the finalisation of the transition to the Web Services Framework leading to increased application development, there are a number of particularly interesting other possibilities. Firstly, the Grid will surely come into its own as more devices become attached, including devices

[10] SuperJanet 5, due in 2006

[11] `http://www.research-councils.ac.uk/escience`

[12] `http://www.ipg.nasa.gov`

that one perhaps wouldn't immediately think of as connection candidates. From a user's perspective it is this, alongside improved interfaces to access Grid resources such as web portals, that will make the biggest difference. From a system administrators' point-of-view, they must very much "bridge technological, political and social boundaries" [1] and IBM's *Autonomic Computing* should ease the management of such large systems through autonomous adaption of components to the demands being placed upon them [53]. From a developer's viewpoint, creating and debugging Grid applications is the key factor in accessing the capabilities of Grid middleware. Lee and Talia [54] summarise the key issues here and discusses various solution models, including shared-state models (e.g. JavaSpace), message-passing models (e.g. MPI), remote procedure call methods (e.g. Java RMI) and hybrids of the three (e.g. JXTA). The one thing that appears certain is that as networks and Grids continue to grow, no single solution will do and ideas from many projects will be needed.

2.3 High Performance Visualization over the Grid

Ever since the first high performance computing resources became available, scientists have needed to visualize their data through graphics, for example, Figure 2.10 shows some of the first use of visualization to take place at the University of Wales, Bangor. In those early days, line plots evolved into vector graphics libraries that used the high performance (particularly in floating point operations) of machines such as the Cray-1 [13]. The switch to raster graphics that occurred brought about the first instance of dedicated hardware being used for interactive visualization of a computation running on a Cray being visualized on a Gould system [55]. Further on, as processors increased in speed and number within HPC machines, parallel polygon rendering was given significantly more attention. Parallelisation of polygonal rendering algorithms has very much lead us to where we are today with HPV using miscellaneous HPC facilities. The co-development of high performance networking has provided the means for connecting HPC facilities with visualization facilities and so, as with the rest of the computing field, high performance visualization has become increasingly based around networks and distributed systems. However, a scientific visualization is usually characterisable as either being visualized on a user workstation after transferring the results of processing, or as visualization being performed on a server and streamed over the network. Both of these approaches lack the flexibility that the Grid aims to provide, and introduce delays and compromises that are far from ideal. That trend continues today with the very latest technology bringing forth and enabling Grid environments in which visualization will

[13] Wikipedia presents a particularly good detailing and discussion of the Cray-1 at `http://en.wikipedia.org/wiki/Cray-1`

play an important part.

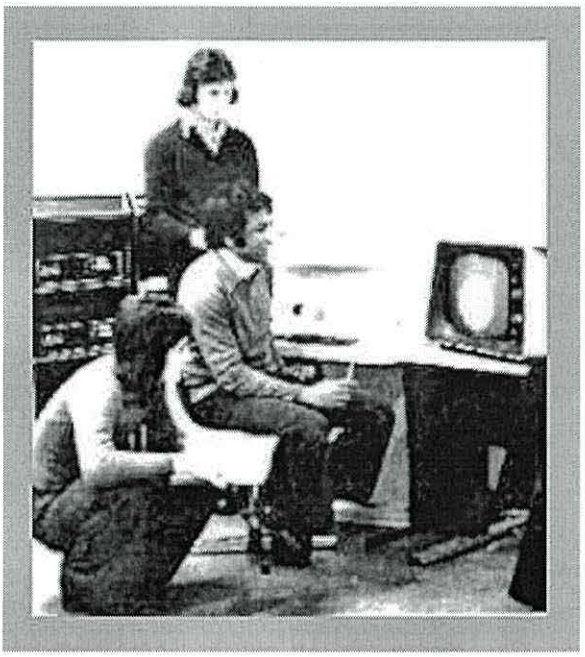


Figure 2.10: Early computer graphics research at Bangor, with Dr Jan Abas at the console, in 1977-78

The pioneering projects discussed below have discovered many implications and issues surrounding the prospect of using a Grid for visualization. It has become accepted that Grids that are built may not be a "one size fits all" scenario. At the very least, sub-grids will be structured in substantially different ways depending on how their resources are to be utilised. Visualization is one incarnation that has quite specific requirements in structure, alongside other examples such as large data platforms (e.g. CERN Large Hadron Collider [14]). Issues that arise in particular in visualization applications include real-time simulation delays, security of data, synchronization of distributed resources, interactive latencies and overcoming the heterogeneity of systems[15]. User interaction with visualization tasks is taken as a given. This has effects on the frame rates required, but does not fit in easily with the concept of virtualization of resources, as in the Grid. Network latencies provide another challenge in that the round-trip time may vary as the network changes end-to-end routes dynamically. Sophisticated components are therefore required to allow interactive

[14] `http://lhc.web.cern.ch/lhc`

[15] The heterogeneous nature of computing resources was highlighted by Benoist, Hewitt and John in [56], and tackled through the use of CORBA in the provision of a standardized network for the creation of heterogeneous virtual environments. Although not strictly Grid work, the similarities in some areas are notable, particularly in regard to harnessing the power of different machines in work for which they are individually optimized.

elements to be reflected back from the user to the various processes running within the Grid.

2.3.1 Griz

Griz is best described as a demonstration of remote rendering over a very high performance network infrastructure. It is the natural combination of a number of technologies discussed above. Utilising so-called "Lambda" network technologies such as WDM (Wave Division Multiplexing) and DWDM (Dense WDM) to allow optical fibres to carry seemingly infinite data alongside commodity PC clusters, Griz adopts techniques from cluster computing in order to drive high-resolution, remote screens over great distances. As with other work, the TCP reliable data transport protocol is found to be unsuitable for this for two reasons. Firstly, the latency introduced by having to do a full round trip for packet acknowledgement and secondly, because the loss of even only a single packet can have a detrimental effect on the performance of a high speed network (10GB/Sec) for many hours [57]. Utilization of the unreliable UDP is free of such problems and allows all the available bandwidth to be harnessed, within CPU bounded limits. Griz made use of the QUANTA networking toolkit [16] to achieve high-performance networking in order to transport data from a rendering cluster in Chicago to a display in Amsterdam. The rendering cluster utilised technology similar to that of WireGL to run a single application across four graphics pipes. The pixels rendered were read back from the framebuffers using *glReadPixels* and reasonable performance of more than 51Hz was found to be achievable on fairly modest rendering hardware [58]. These pixels were then transmitted over the high-performance network, occupying up to 1.5Gbps for an image of total resolution (four tiles) 1600x1200 at interactive frame rates (16-23fps). The Griz authors observe some interesting effects in bandwidth utilization as resolution increases, most likely explainable by read back inefficiency. Griz's demonstration of interactive remote rendering utilising high performance networks establishes an important precedent that such systems are possible and realistic.

2.3.2 Cactus from LBL

Cactus is a project a little out of place here. It is actually a problem-solving environment that consists of many modules. However, Cactus is designed for easy parallel work and collaborative development, and integration with Globus and the Grid has reached a developed

[16] `http://www.evl.uic.edu/cavern/quanta`

stage. Additionally, Cactus provides for code development in a number of variants of C and Fortran, and for cross-platform code development and execution. Cactus produces an abstracted layer above the Grid middleware through its provision of 'thorns' (Cactus components) that implement different execution methodologies on the Grid [59], for example an MPI thorn. I/O is handled in basically the same way. Cactus can also link to visualization products such as Amira [17] to produce some stunning graphics. Using Cactus' socket-based data streaming capabilities, remote visualization of live computations can be made. The collaborative environment is enabled through the ability of Cactus to send a data stream to multiple clients simultaneously [60]. A number of additional tools have been developed for Cactus in the Grid for purposes including checkpointing of distributed simulations and remote application steering, as well as in developing portals to access Grid services.

2.3.3 RealityGrid

The RealityGrid project is aimed at facilitating computational studies of complex condensed-matter systems. By improving scientists' capability to interact with the massively expensive hardware they use for this work, RealityGrid aims to enhance scientific productivity. Instead of an off-line, batch controlled computation, RealityGrid enables a scientist to interact with their simulation whilst it is running through computational steering. In order to understand how to interact with it, the scientist views a visualization targeted to the hardware they have available, from workstation to PDA. RealityGrid applications are built in a 3 component structure of simulation, visualization and steering client. The steering framework utilises the third generation Grid standard OGSI. A small steering service sits between the simulation and the steering client in order to provide the required Grid service. The discovery of the steering service by the steering client is achieved by having the service register its presence within a Registry that the client later consults [61]. As the communication requirements between the client and the simulation are not intense, this traffic can travel as Simple Object Access Protocol (SOAP) via HTTP. However, the intensity between the simulation and the visualization is far greater and hence requires high-performance transport, such as disk based I/O or the socket based Globus I/O [62]. The TeraGyroid Experiment showcased RealityGrid at SuperComputing 2003. This used GRAM and Grid FTP to launch the various components of RealityGrid and tie together high-performance resources on both sides of the Atlantic, including those at CSAR and Argonne. TeraGyroid utilized the AccessGrid to multicast the generated video streams between the component sites as rendered through Chromium and transmitted via the FLXmitter library. Alongside SGI OpenGL VizServer (see

[17] `http://www.amiravis.com`

later) and VNC, this was transmitted over standard production networks. The specially provided high-performance networks procured for the experiment were used to transmit the datasets between the sites. TeraGyroid also demonstrated some of the difficulties to be experienced in requiring sites in one Grid scheme to trust the certificate authorities of others and in attaining the theoretical levels of performance of high-performance networks. However, all the many problems were overcome by "heroic efforts" to provide a telling demonstration of future possibilities.

2.3.4 GVK

The Grid Visualization Kernel (GVK) [63] is a middleware extension that allows the interconnection of the various parts of a scientific visualization - data sources, simulation processes and visualization clients. GVK is capable of dynamically changing this visualization pipeline without user knowledge, according to changing network conditions. By performing various rendering techniques between the computation server and visualization client, GVK can dynamically change the nature of the visualization. To construct this setup, GVK is implemented as input and output modules for a number of visualization packages including OpenDX and AVS. The fact that each of these packages is based on data-flow and a custom API makes it possible for these extension modules to provide the necessary capability. To utilise GVK, it is only necessary to use an input interface at the simulation server and an output interface at the visualization client. The user also specifies the characteristics of the visualization (e.g. resolution, colour depth, etc) and some details on the nature of what is being visualized. GVK uses the possible pre-defined states to optimize its pipeline. GVK then schedules and creates the pipeline with appropriate components in what it defines as sensible places, traditionally one of:

- Client: Filtering, Visualization and Rendering.
- Server: Filtering
  Client: Visualization and Rendering.
- Server: Filtering and Visualization.
  Client: Rendering.
- Server: Filtering, Visualization and Rendering.

By adapting this balancing to network conditions, GVK becomes adaptive to the real world. However, GVK can also place more advanced function into the nodes it uses [64], including

level-of-detail filtering and occlusion-culling, in order to further optimize to the network bandwidth and latency. GVK utilises a typical Grid approach of providing middleware and hiding the maximum amount of low-level functionality from the user as possible. By manipulating the graphics pipeline and applying traditionally non-distributed techniques behind the scenes, GVK adjusts to changing network conditions with minimal effect for the user. This is a novel and worthwhile approach.

2.3.5 RAVE

The Resource Aware Visualization Environment (RAVE) [65] is an on-going research project at Cardiff University and the Welsh e-Science Centre aimed at enabling collaborative sharing of resources on the Grid. RAVE supports a wide variety of devices from PDAs to high-end visualization systems (such as an SGI Onyx or Prism) and can place different parts of the rendering process in different places to suit resource distribution. This enables multiple users to collaborate in work from wherever they are using whatever interface is available to them - for example, the user in the office with the large visualization machine may utilize a large wall-style display and may be collaborating with a user who is on a PDA where the rendering is actually done by a remote server and the pixel-image is transmitted for display on the PDA. RAVE implements a central data server for the distribution of necessary data to users, two clients for the users (one active doing rendering itself, and one 'thin' using remote rendering) and a render service for performing remote rendering which operates in the background and does not interfere with the console of the executing machine [66]. RAVE is based on the Web Services Framework Grid standard and is being integrated with a selection of applications, including molecular dynamics and galaxy simulation.

2.3.6 SGI Visualization Area Network and Media Fusion

The SGI Visualization Area Network (VAN) is another idea aimed at removing the physical barriers to collaborative and remote visualization. VAN aims to makes the power of large SGI visualization servers available to individuals and teams in disparate locations. It uses OpenGL VizServer to provide the rendering power of the servers back to the users. This effectively works in the same way as similar products, by transporting the 2D pixel data for a rendered image from the VizServer server to be displayed on a VizServer client, perhaps running on a far less powerful desktop system or in a remote projection theatre. VizServer can make use of the multiple graphics pipelines in such a high-end SGI box in a separate

or coupled manner. VAN itself represents a holistic approach by encompassing storage, compute, networking and visualization hardware alongside the software element.

Media Fusion represents the next step on from how the computing field has been in the past to a future with different balances in hardware and networks. The Media Fusion Environment [67] represents a 3-stage pipeline of ingestion, fusion and distribution that treats the pixel as the base data element and handles multiple pixel streams simultaneously. The ingestion stage consumes the multiple pixel streams (for example from a local application, over a VAN or some form of video stream) and buffers it in memory. The fusion stage is then capable of compositing pixels together and processes the streams appropriately to enable multipipe rendering and other filtering of the data. Both ingestion and fusion require multiple high-performance CPUs and ingestion requires sophisticated input capabilities. The distribution stage is concerned with passing out appropriate fused media streams via local or remote rendering to provide an Immersive Virtual Environment to collaborating users. A visualization 'portal' has also been proposed as a future development of the Media Fusion idea in which heterogeneous user terminals utilise the abilities provided by Media Fusion servers in order to harness the power of the full system - storage, compute and network - which sounds like a Grid resource description.

2.4 Critical Appraisal

In assessing the Grid Visualization solutions and contributions detailed above, the four most prevalent questions are: why was this project attempted?, how was it attempted?, what were the results?, and what does this mean? Many of these details have been covered above, but we draw them together here in an appraisal and comparison of the different solutions.

Why? The solutions detailed above address various different areas that tie in to Grid and Visualization work. Griz and the SGI VAN/Media Fusion architecture both demonstrate high-performance visualization and express future possibilities. However, they are bespoke solutions and have been designed independently of the Grid initiative. This is in contrast to the other four projects identified, which are to some extent Grid based. We see that all four are broadly aimed at scientific visualization, principally through their use alongside traditional Problem Solving Environments. Cactus and GVK are both used clearly in this way, whereas RealityGrid and RAVE are specifically aimed at Grid based visualization. Additionally, as a result of their focus on the Grid, RealityGrid and RAVE both offer native

collaboration techniques, whereas such techniques are separate to the Grid and Visualization components of GVK and Cactus.

How? The origins of each of the above projects is seen clearly in how they have been implemented. Specialised (and thus, expensive and not generally available) networks of various scales are used to enable the work, over the large scale with Griz, RealityGrid and RAVE, but over a smaller scale with the SGI VAN/Media Fusion. Cluster-based rendering is enabled by Griz and RAVE in particular, although parallel rendering is a significant part of the SGI solution. Use of Grid standards is complete in the RealityGrid and RAVE projects, and an additional part of the Cactus thorn that provides such capabilities as part of the Cactus 'pipeline'. The now previous-generation of OGSI, GRAM, GridFTP and Globus IO are used for RealityGrid, whereas current Grid technolgies WSRF and UDDI are used in RAVE. However, it is worth noting that momentum may be moving away from UDDI at this time [18].

What? The results of these projects are highly-relevant to the future of Visualization and Grid software, and the interaction thereof. Griz finds that TCP is poor for supporting the level of data implicit in remote visualization over large distances, but also that high-performance networks make trans-continental interactive visualization a realistic proposition. RealityGrid demonstrates this principle through the use of dedicated networks to provide a distributed scientific visualization with computational steering. Grid protocols are found to be usable in this, with certain traffic types suited to certain protocols. RAVE shows that a heterogeneous architecture of systems can be used in providing the visualization, and that such work can be carried out using conventional networks. However, the performance demonstrated by RAVE on the whole is not good enough for interactive applications. GVK tackles performance issues through dynamic movement of pipeline components behind the scenes, but does not do so over the same levels of distribution. Cactus and GVK share some similarity in that they are implementing Grid Visualization as a plugin for a pre-existing scientific environment.

So? Interpreting the ideas, work and results of the above six projects shows great achievements and some shortcomings in the areas targetted in this thesis, primarily due to the different focus of the various projects. Considering use of and suitability for a Grid environment, it is obvious that Cactus, RealityGrid and RAVE integrate well with existing Grid standards. However, the rapid advance of Grid protocols and standard does suggest

[18] See the news story at `http://www.techworld.com/applications/news/index.cfm?NewsID=5030`, which reports that Microsoft, IBM, and SAP are discontinuing the UDDI Business Registry (UBR) project for Web services in early 2006.

that parts of these solutions will require frequent updating. GVK, Griz and the SGI Media Fusion architecture are not aimed at the Grid in the same way, and so do not use standard Grid middleware. The use of WSRF by RAVE fits in with the latest generation of Grid standards. Another key area in this thesis is support for real-time, interactive visualization. Whilst we see that the six projects are considerate of this, it is not key to their operation as their primary focus is on the science of the visualized simulation and making it available in the relevant form. Support for existing, standards-based applications is also limited to current work. SGI Media Fusion, as a development and utilization of SGI VAN, obviously offers compliance with and support for OpenGL based applications. Griz utilizes a WireGL-like system for its parallel rendering, implying its support for standard graphics APIs. However, RealityGrid, Cactus, GVK and Rave do not provide support for such existing standards-based applications in a Grid environment, instead supporting either custom written simulations or simply 'tagging-on' as a component of an existing system. On the whole, we believe that whilst these projects offer excellent functionality, there are clear areas lacking and a number of non-scalable solutions implied (for example with non-commodity network connections). However, we certainly can imagine that a future all-encompassing visualization system based on Grid technologies would exhibit characteristics from all the projects. It is for this reason that key parts of existing research are highly relevant to our work in this thesis, whereas our novel approach produces many new challenges that, as we've seen, have not been fully addressed.

2.5 Conclusions

There are two key fields that feed into this work. Firstly, the field of high-performance visualization, in which an increasing amount of attention is being given to the performance of commodity graphics accelerators. Such devices give such levels of performance that they have been used on the small-scale in desktop PCs, and on the large-scale in parallel within high-performance SMP machines. When put alongside the increased performance seen in Local Area Networks, with 100Mb/Sec as a minimum and Gigabit Ethernet being highly affordable and increasingly widespread, the concept of graphics clusters becomes very enticing, as evidenced by the work now going on in this area. The second field is that of Grid research. The Grid clearly represents the connected future and an architecture to provide services over future networks. Along with the ever-improving national and international networks, it will be possible to provision high-bandwidth services over increasingly distributed areas.

Hence, is it possible to bring these two areas together to provide the power of parallel graphics in an easy-to-use and, in the future, increasingly distributed environment? As discussed here, there are a number of projects tackling such an idea. However, these tend to focus on delivery of new applications in a way that is well-suited to the Grid environment. We believe that the Grid is not the revolution that this implies but is actually part of the evolution, and as such it needs to support current ways of working well. Hence, we see a need for research into running real-time, interactive, standards-based (OpenGL) visualizations over parallel systems, and doing so with the ease-of-use that a Grid environment should provide. Effectively, we want to make a Grid visualization system and study such a system in a conventional (rather than purpose-built, high-performance) environment. There is no point in reinventing the wheel, so we have chosen to use Chromium as the visualization element and pair it with Globus, the standard Grid middleware, for this project.

Chapter 3

The jgViz Grid Information Model

Distributed systems on the scale of Computational Grids present many unique problems and challenges. Such systems involve numerous different services running on a wide variety of machines in distant and disparate locations. In order to utilize them, it is necessary to be able to both find them and find out about them. We need some structure for discovering available systems that provides a mechanism for querying the current status and services offered by that system and is be able to react dynamically to any change in status of the services. Our *Java-enabled Grid Visualization System (jgViz)* represents our work on grid visualization, of which such an information model is a key part.

The jgViz information model utilises available Grid capabilities to provide a scalable and capable system in as efficient a way as possible. The jgViz Grid Information model is one of the key contributions to the field resulting from this thesis. jgViz discovers available resources and combines them with various inputs to create a distributed graphics pipeline on the Grid. jgViz consists of two primary components to achieve this. Firstly, a server component runs on all the machines that are making graphics resources available, and secondly, a client that the user interacts with in order to find and utilise the available resources. However, a vital component is the language in which the components converse. The exchange of relevant and current information obviously plays a defining role in how well the end-product turns out.

3.1 The Language of Visualization

It has long been acknowledged that it is difficult to describe a general-purpose visualization system [68]. Products such as IRIS Explorer and OpenDX provide the ability to link together modular graphical components, but do so within a relatively strict set of interfaces and a single project. We need to be able to express both the available visualization resources on a machine and the resource requirements of a job. In addition to providing such information, 'live' data on current utilization and other changeable values are needed in order to perform time-sensitive scheduling. We therefore require some sort of active work on those machines that are advertising capabilities to provide this information.

There are many options in terms of both the dictionary and the grammar of the language used to describe visualization resources. The gViz project [69] studied the representation of such data in XML format at three levels: the conceptual layer that loosely explains the structure for visualization and collaboration; the logical layer, at which the concepts are mapped to software entities; and the physical layer, at which the logical structure is mapped to a grid environment. The *skml* language [70] this forms is used in the gViz extension to IRIS Explorer that sets up a pipeline on Grid resources. However, stemming from Molnar et al [71] and the splitting of the graphics pipeline into separate sections for parallelisation, our language needs to focus on providing the elements to create parallel rendering in a distributed environment and not the contained components of a package such as IRIS Explorer. All the work that has gone into parallel visualization research and the increasing number of commercial solutions gives us options and shows us that we shouldn't reinvent where unnecessary. The commercial products mentioned previously are principally very proprietary and more concerned with hardware than software. However, there are several interesting research projects. *AnyGL*, from Yang et al, presents a system that offers some novel features (such as global sharing of textures and display lists) and performs well [20], but is also not open source and provides a pipeline structure that is difficult to distribute effectively. Winkelholz and Alexander present a system in [21] that is focused on virtual environments through the Virtual Reality Modelling Language (VRML) and so not as versatile and general-purpose as we would like. Vo et al [72] and Bethel et al [73] both present particularly interesting work on parallel scene-graph visualization with consideration of the memory-based distribution problems presented involved. Whilst Bethel et al make use of Chromium for their work, the targeting of scene-graphs puts both of these pieces of work into too specialized a category for this research. Chromium on its own, however, presents a good balance of the many aspects we require. Chromium operates primarily in the real-time space of OpenGL based applications and provides performance known to be on

an equal footing with other such systems [18] that provide less functionality. Additionally, the general-purpose nature of the Chromium framework, with its different classes of 'node', provides a dynamic and highly-customisable structure that we can utilise.

Chromium utilises its own syntax in configuring the nodes and stream processing units that comprise a parallel graphics pipeline, and we therefore use this as a base to work from. The two fundamental configuration elements within a Chromium session are nodes and Stream Processing Units (SPUs). This concept maps straight into our Grid based structure by allowing machines to advertise themselves as nodes that are capable of providing service with a set list of SPUs. Also, Chromium's modular architecture means that it is easy to implement extensions that may improve operation in a distributed environment (e.g. compression of network streams). With the exception of the finer points of launching a Chromium session, the designation of Chromium nodes as either clients or servers is not relevant. However, the launching and structure of a Chromium session clearly implies a relevant classification of nodes being either mothership, application or server nodes. For the purposes of launching on the Grid, jgViz treats these all as different nodes whether or not they are on the same physical machine. The mothership node is that on which the Chromium mothership runs and which all other nodes need to talk to initially to discover their function. The application node runs the application in question and fakes it to use Chromium's libgl, the output of which is processed and sent over the network as necessary. The process of 'faking' the application involves setting the *LD_LIBRARY_PATH* system variable to cause the run-time linker to preload the Chromium libgl in place of the system libgl so that OpenGL calls are intercepted and fed to the Chromium SPU chain instead of the local graphics hardware. Chromium's SPUs will make use of the system libgl at a later stage in order to harness the power of that hardware. It is therefore a good idea for this node to be 'near' (in network terms) to the servers it is feeding with graphical data. These server nodes are the slaves that do the actual rendering and thus the machines that we are really interested in automating the discovery and launch of, as there may be many of them.

Chromium pipelines are 'programmed' as Python scripts extended from the mothership base to represent a directed acyclic graph. This use of a scripting language to setup the graphics pipeline allows Chromium startup scripts to contain functional sections involving loops and decision making constructs. Such a mechanism of modelling the pipeline components as objects that need to be instantiated, configured and then called in a certain way, provides a diverse configuration ability that we can make use of in constructing distributed graphics pipelines. A simple annotated example Python script is shown in figure 3.1.

As the placement and configuration of the mothership and application nodes is subject to

```
import sys
sys.path.append( '../server' )
from mothership import *

if len(sys.argv) > 3 or len(sys.argv) < 2:
    print 'Usage: %s <demo> [spu]' % sys.argv[0]
    sys.exit(-1)

demo = sys.argv[1]

if len(sys.argv) == 3:
    clientspuname = sys.argv[2]
else:
    clientspuname = 'pack'

server_spu = SPU( 'render' )
client_spu = SPU( clientspuname )

W = 500
H = 500

server_spu.Conf( 'window_geometry', [100, 100, W, H] )
server_spu.Conf( 'swap_master_url', "" )
server_node = CRNetworkNode( )
server_node.AddSPU( server_spu )

if (clientspuname == 'tilesort' ):
    server_node.AddTile( 0, 0, W, H )

client_node = CRApplicationNode( )
client_node.AddSPU( client_spu )
client_spu.AddServer( server_node, 'tcpip' )
client_node.SetApplication( demo )
client_node.StartDir( crbindir )

cr = CR()
cr.MTU( 1024*1024 )
cr.AddNode( client_node )
cr.AddNode( server_node )
cr.Go()
```

Inherit the Chromium mothership

Check job launch and setup as appropriate

Server (rendering) node

Application node and details

Create an actual Chromium session (mothership)

Figure 3.1: Example Chromium configuration script (for remote/tiled rendering)

different requirements from those of server nodes, these are configured separately by the user in jgViz. There are three types of data that jgViz will deal with for server nodes. Firstly, in order to construct a distributed pipeline, jgViz needs to discover the information that is required to configure each server component thereof. This is termed the *Chromium Configuration Data*. Secondly, for the purposes of the construction of the pipeline by jgViz, other information is needed to cover details such as position in a tiled display. This data is termed the *Additional Server Data*. Third and final is information relating to external factors such as machine load, as used for scheduling by jgViz. This is termed the *Server Status Data*.

3.2 Server Nodes

The server component of jgViz is that which executes on those machines making themselves available as visualization resources. It acts to advertise both the presence and capabilities of the machine. As shown in figure 3.2, there are three types of data that need to come from such machines. This data is sourced from multiple places.

Figure 3.3 shows that there are several components involved in providing the Grid information on a server node. The principal component is an OpenLDAP server (a GRIS) that is configured to read a core LDAP schema and additional schema files as necessary. These schemas form the server's language (an LDAP tree hierarchy). To populate the individual sub-trees defined in the LDAP schemas (for example, the viz sub-tree), the OpenLDAP server calls a number of executable files (so-called *Information Providers*) that return LDAP Data Interchange Format (LDIF) data on the standard output stream. This LDIF data is 'consumed' by the LDAP server (called *slapd*), which it then makes available for searching. The jgViz information provider is a simple C program that outputs the LDIF data for each of the configurations setup by the administrator. A C-compiled executable is used here due to the lightweight overhead of launch and execution. Java, for example, has too much overhead in terms of creating the Java virtual machine, particularly for such a process that is going to be launched very frequently but will only execute for short periods. Any data that the server sees that is not within the language it knows of is ignored. Hence, when jgViz related data is passed to a non-jgViz enabled server, it does no damage and is silently dropped. The output format is shown below:

```
dn: Mds-Software-Deployment=jgviz 0, Mds-Software-deployment=viz,
    Mds-Host-hn=node01.sees.bangor.ac.uk, Mds-Vo-name=local, o=grid
```

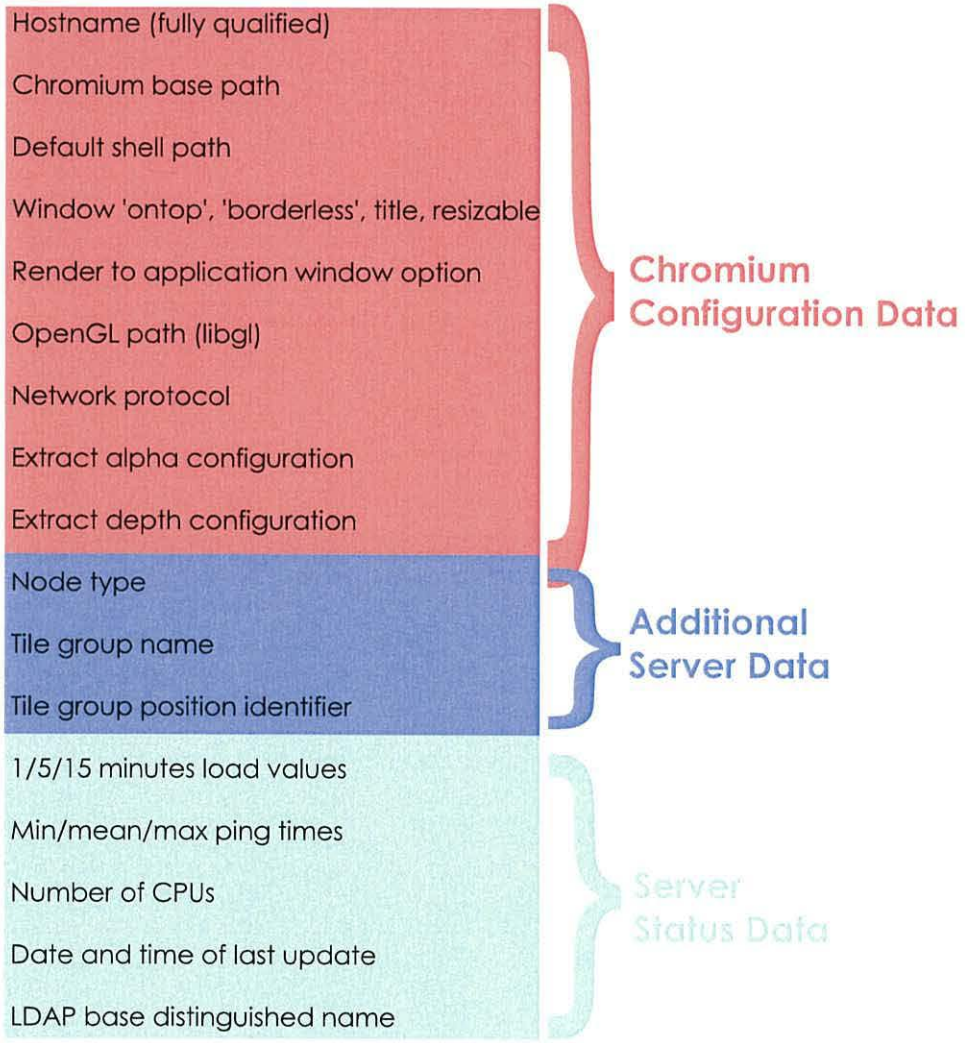


Figure 3.2: Information components of a jgViz visualization server node

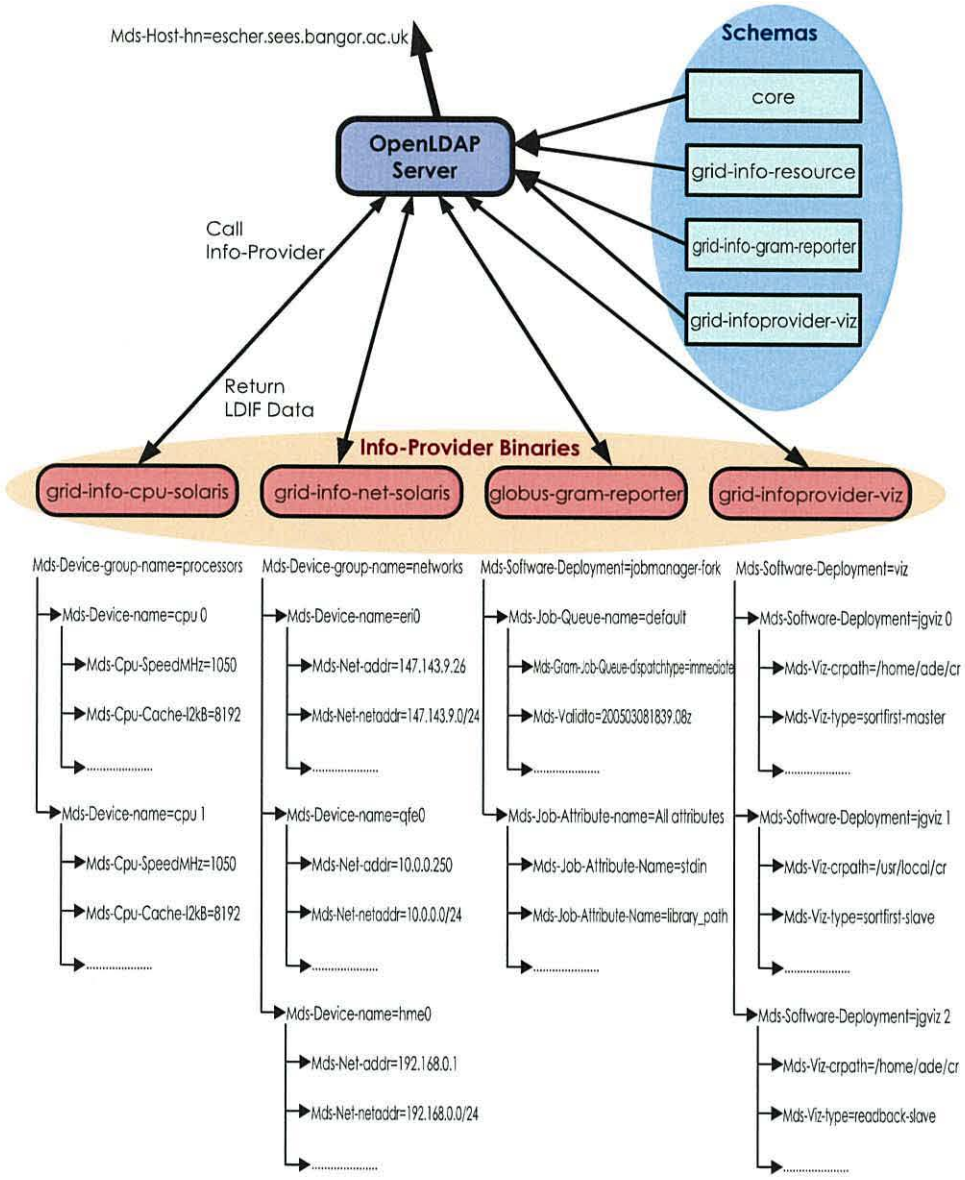


Figure 3.3: Server-side structure of jgViz information components alongside LDAP hierarchy

```
objectclass: MdsVizData
Mds-Viz-FQDN: node01.sees.bangor.ac.uk
Mds-Viz-Date: Wed Mar  9 16:30:26 GMT 2005
Mds-Viz-Type: sortfirst-slave
Mds-Viz-OnTop: 1
Mds-Viz-Borderless: 1
Mds-Viz-GLPath: /usr/lib/tls
Mds-Viz-Protocol: tcpip
Mds-Viz-CrPath: /home/ade/cr
Mds-Viz-Path: /bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin
Mds-Viz-WinTitle: cr_tile_render_node01
Mds-Viz-ExtractAlpha: 0
Mds-Viz-ExtractDepth: 0
Mds-Viz-RenderToAppWindow: 0
Mds-Viz-Resizable: 0
Mds-Viz-Load-One: 0.00
Mds-Viz-Load-Five: 0.00
Mds-Viz-Load-Fifteen: 0.00
Mds-Viz-TileGroup-Name: pipe1@bangor
Mds-Viz-TileGroup-Number: 1
Mds-Viz-Processor-Quantity: 1
```

The configurations that the administrator of a potential server wishes to make available are defined in a number of uncomplicated text files. The jgViz information provider executable will read configuration files and advertise configurations from 0 to n, so the administrator can configure as many setups as they wish. Additionally, the information provider adds certain live data for items such as *Mds-Viz-Load-One*. When multiple configurations are present, they are advertised in a hierarchy as shown in figure 3.3. The tree structure splits into n branches, each representing a different available configuration for that server. In this way, the administrator retains tight control over how their machine is used, as it may be important to enforce control over its use. Example problems that may otherwise occur include hijacking of a server that is meant to be used in a tiled display group and use of a non-compatible OpenGL library.

If the GRIS server that is advertising a jgViz setup is configured so as to upload to a GIIS server then, providing the GIIS server knows the jgViz LDAP schema, the GIIS becomes an aggregator of a number of different jgViz machines available. This aggregation works at all

higher layers of the index hierarchy through the Grid Information Services standard. The *objectclass* attribute also shown above is a feature of the LDAP format. It defines a 'type' for the data record it applies to. We have created *MdsVizData* as the objectclass for jgViz data records.

3.3 Client

The jgViz client takes the form of a graphical user interface written in the Java language and using the Commodity Grid Toolkit. This allows it to make use of the Grid capabilities as required. It is necessary that the machine running the client is a 'grid-enabled' machine. The processes of resource discovery and configuration are carried out together using the Grid Information Services structure of GRIS and GIIS servers. jgViz allows the user to find the available configurations on a single machine (via a GRIS) or on a collection of machines (via a GIIS), or pick out an individual configuration from a single machine if required.

In order to do this, the jgViz client communicates with the LDAP server and queries for all objects of objectclass 'MdsVizData'. This is termed the search filter. The jgViz client then recurses through the search results and extracts the individual node entries. The client maintains a Java Vector of the resources found which then represents available server nodes of all types.

As previously mentioned, it is most likely that the jgViz user will have some idea of where they want to run their application (often *localhost*) and where they want to set up their Chromium mothership (possibly *localhost*). For this reason, jgViz allows the settings involved in both of these instances to be user defined, with sensible defaults preset. Depending on the nature of the application involved, it may need to run on a particular high-performance compute facility or with particular, unique options. Also possible is that the mothership and/or application may need to be started on particular machines in order to be available to a number of private networks that are not routable otherwise.

Once all the available resources have been found, the user specifies the type of graphics pipeline they want to use, how many server nodes they wish to use, etc. and scheduling takes place to pick the best resources. This requires more up-to-date data than is necessarily available in the last LDAP lookup we made as well as other additional information. We therefore carry out some active processing both inside and outside of the GIS in order to obtain this data.

There are two pieces of dynamic data involved here. Firstly, the load of the target machine relative to its capability, thus representing how busy it is, which is obtained by getting an up-to-date value from the GRIS server directly on the target server. Secondly, we also need to establish data on the network connections involved, for which we send a number of *ICMP ECHO (ping)* packets to the target machine and utilise the output accordingly.

The output of the scheduling process is another vector, this time containing the chosen nodes to use for the graphics pipeline. This is passed to another part of the jgViz client in order to generate a Chromium python configuration script. Depending on the type of graphics pipeline to be created, the vector will take one of two forms. The data passed takes a slightly different form depending on whether a readback or tiled pipeline is used (see chapter 4). The configuration building algorithm implements the necessary subset of the complex Chromium configuration language in order to produce our distributed graphics pipelines. This is then passed on to the runtime component of the client.

3.4 An Example

The best way to understand many of the details explained in this chapter is through a real-world example. Here, we present just such an example - in this case, the use of two simple workstations for all parts of the pipeline to produce a two display tiled output. There are, however, three machines involved as a third grid node runs the jgViz client in addition to the main two grid nodes.

3.4.1 Server Configurations

The jgViz information provider is called by the OpenLDAP backend. To produce the required LDIF data, the provider reads as many configuration files as it can find, one for each potential configuration the administrator of the said machine decides to make available. This file needs to be created with a simple syntax. In our simple example, the two jgViz servers offer the configuration shown in figure 3.4.

So, we see in figure 3.4 that the machine *node02.sees.bangor.ac.uk* makes available three configurations including one as a tiled display slave. *node01.sees.bangor.ac.uk* also makes available two configurations. These two machines are both part of the Bangor sub-grid of the UK e-Science Grid and so both pass their data to the local Bangor GIIS indexing

node02 jgViz configurations

gridviz.conf.0
fqdn node02.sees.bangor.ac.uk
type sortfirst-slave
ontop 1
borderless 1
glpath /usr/lib/tls
protocol tcpip
crpath /home/ade/cr
path /bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin
wintitle cr_tile_render_node02
extractalpha 0
extractdepth 0
rendertoappwindow 0
resizable 0
tilegroupname pipe1@bangor
tilegroupnumber 0

gridviz.conf.1
fqdn node02.sees.bangor.ac.uk
type readback-slave
ontop 1
borderless 1
glpath /usr/lib/tls
protocol tcpip
crpath /home/ade/cr
path /bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin
wintitle cr_readback_node02
extractalpha 0
extractdepth 0
rendertoappwindow 0
resizable 0
tilegroupname pipe2@bangor
tilegroupnumber 0

gridviz.conf.2
fqdn node02.sees.bangor.ac.uk
type readback-master
ontop 1
borderless 1
glpath /usr/lib/tls
protocol tcpip
crpath /home/ade/cr
path /bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin
wintitle readback_render_node02
extractalpha 0
extractdepth 0
rendertoappwindow 1
resizable 1
tilegroupname none
tilegroupnumber 0

node01 jgViz configurations

gridviz.conf.0
fqdn node01.sees.bangor.ac.uk
type sortfirst-slave
ontop 1
borderless 1
glpath /usr/lib/tls
protocol tcpip
crpath /home/ade/cr
path /bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin
wintitle cr_tile_render_node01
extractalpha 0
extractdepth 0
rendertoappwindow 0
resizable 0
tilegroupname pipe1@bangor
tilegroupnumber 1

gridviz.conf.1
fqdn node01.sees.bangor.ac.uk
type readback-slave
ontop 1
borderless 1
glpath /usr/lib/tls
protocol tcpip
crpath /home/ade/cr
path /bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin
wintitle cr_readback_node02
extractalpha 0
extractdepth 0
rendertoappwindow 0
resizable 0
tilegroupname pipe2@bangor
tilegroupnumber 1

Figure 3.4: Example server configurations

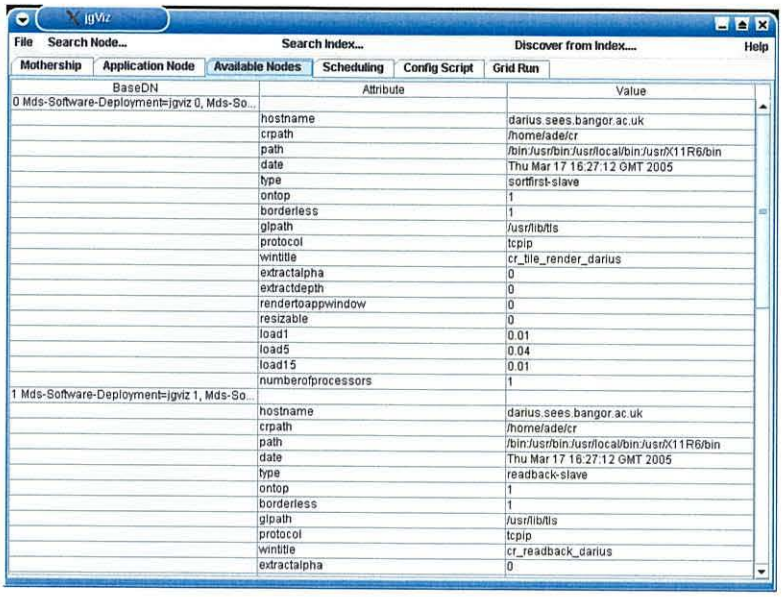


Figure 3.5: jgViz finds available server nodes

server. As these machines have been configured to understand the jgViz LDAP Schema, they make such data available in a searchable manner. As the UK national GIIS server has not been configured to understand it, such data is silently dropped and kept local to Bangor only. The timeout values defined by jgViz for the various pieces of data in the GIIS are short enough (20 seconds) that when a potential server goes away, the data will quickly disappear from the indexes and hence the resource will not be discovered again.

3.4.2 Client Configuration

When the user wishes to make use of a distributed graphics pipeline, they start up the jgViz client and enter (or load from a saved file) the details for the mothership and application nodes. Both come with sensible default values so it will typically only be the hostname that requires entry. In order to discover the graphics resources available, the user simply needs to enter the name of the GIIS for the local site. A search is performed and all available resources are found and located.

Now that the jgViz client has learned and acquired knowledge about all the available nodes, the user progresses to the 'scheduling' section where they specify what type of graphics pipeline they require in terms of both structure and number of server nodes. The user also specifies the output resolution and jgViz then performs the tiling of the resulting scene

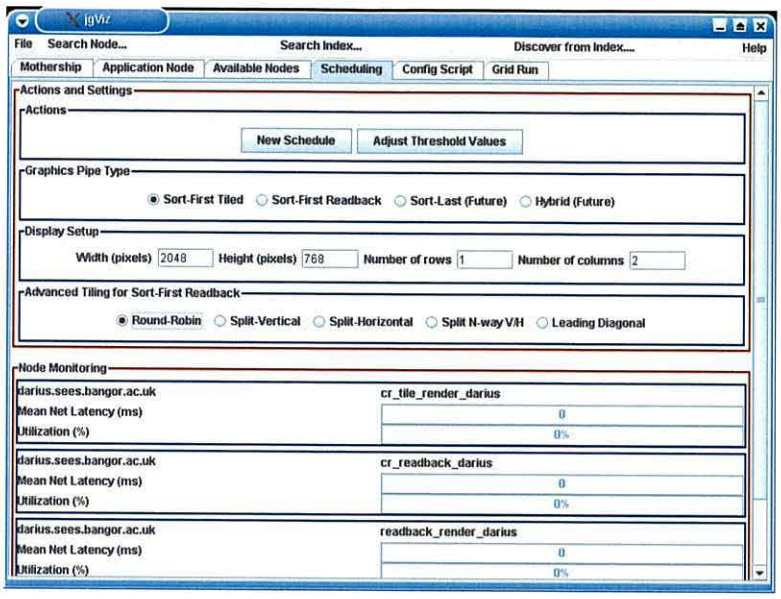


Figure 3.6: jgViz is configured by the user, whilst monitoring available resources in the background

automatically.

Whilst the user is doing this, the client is monitoring the load and network latency of the nodes that it has available so that it has current and consistent data when the user chooses to hit the *Schedule* button to form a pipeline. Once the scheduling process has taken place, the *Configuration Script* tab of the client generates a Chromium configuration script based on the scheduling output. This script is then passed through to the jgViz runtime component.

3.5 Conclusions

The information gained from various resources and used within jgViz follows a linear and transparent path through the server and client components. In chapter 4, we deal with the more involved parts of this process. Figure 3.8 gives a diagrammatic overview of the entire production and consumption of the information, all the way through to the runtime stage. Major information flow during resource discovery is through the Grid Information Services index servers hierarchy with additional data discovered directly from nodes when scheduling amongst them.

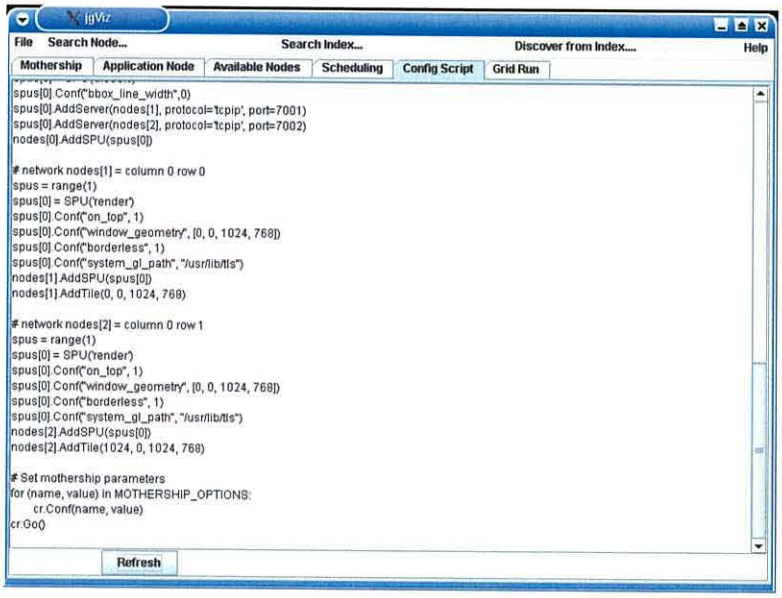


Figure 3.7: jgViz uses the scheduled data to produce a Chromium script

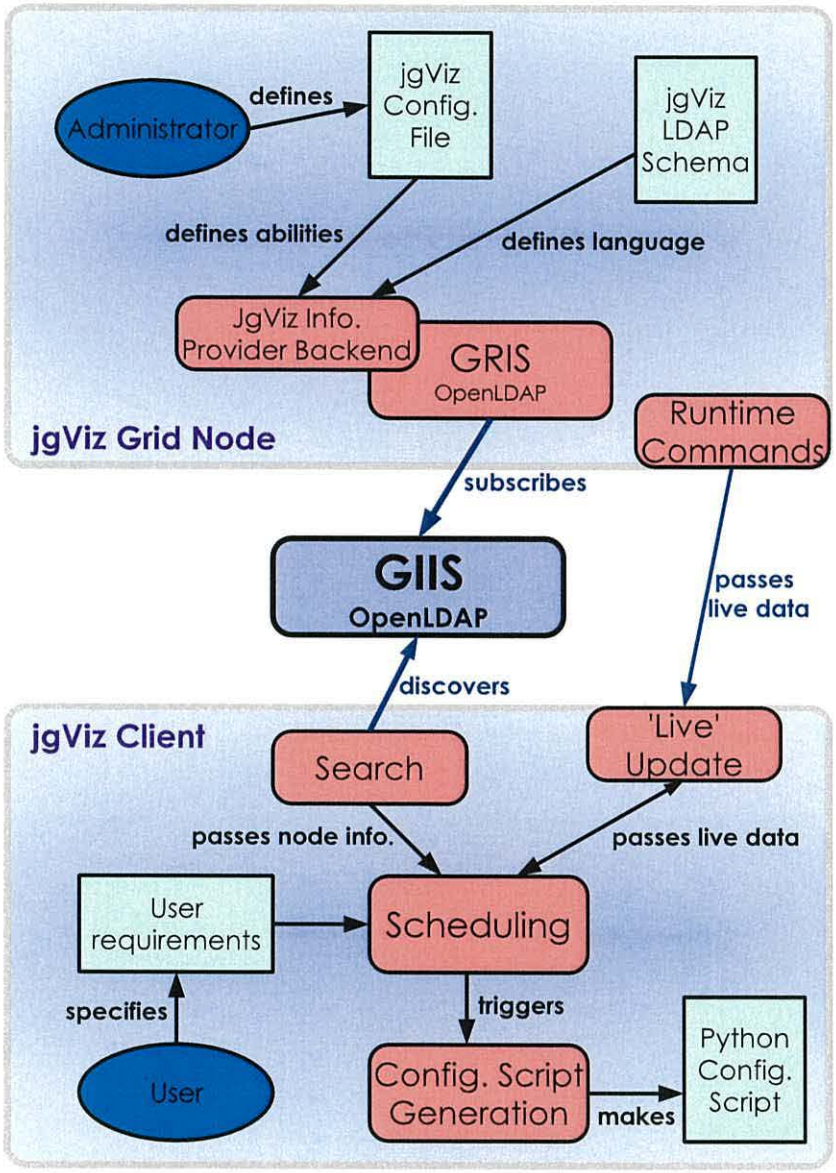


Figure 3.8: Information flow through jgViz

Chapter 4

Scheduling in jgViz

Distributed systems on the scale of Computational Grids present many unique problems and challenges. With the vast number and range of systems that are likely to be available for utilization in a Grid environment, as discovered for us by jgViz's Information Model subcomponents, how do we decide which to use to provide optimal performance? The process of scheduling, or deciding upon a group of resources according to some criteria, tackles this and is the subject of this chapter.

jgViz's scheduling is subject to a number of restrictions and requirements. At its most basic, it must extract a list of suitable resources for use from a larger list of available resources previously discovered. However, this process needs to have some intelligence associated with it. It makes sense to utilise the subset that gives the best performance, and the most sensible way to achieve this in the changeable Grid environment is to assess a set of dynamically changing metrics. Once measurements have been taken, then the available server resources can be subsetted to obtain the best performance subset to use.

4.1 The Scheduling Process

jgViz's scheduling is a multi-stage process that takes in a list of available resources and outputs a list of active resources. Involving the measurement and scoring of resources, the scheduler must make significant (if not substantial) use of the network itself. In jgViz, the scheduling stage also involves the process of building a Chromium configuration script. Figure 4.1 shows the scheduling information flow.

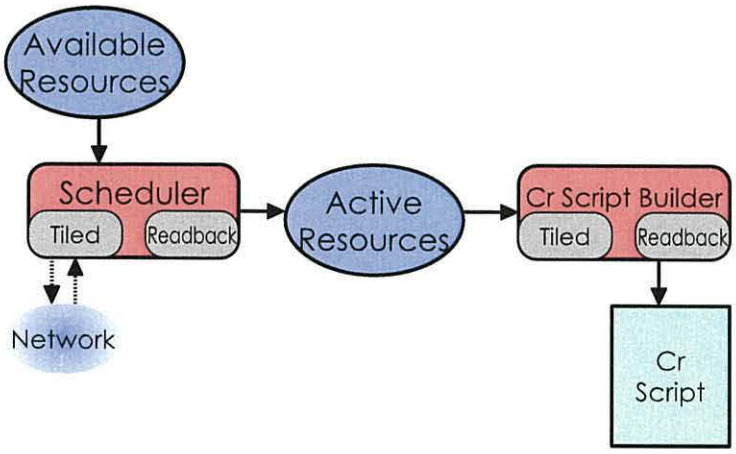


Figure 4.1: Information flow through the jgViz scheduling subsystem.

We now tackle each of these elements in turn, covering the important technical and otherwise issues for each component of the scheduling subsystem.

4.1.1 The Scheduler

The scheduling part of the jgViz system takes place within the client. The output of the resource discovery process is the collection of *Available Nodes*, which are the machines located through the Grid index search. Once the discovery phase has ended (at the user's discretion), the client begins to periodically update the 'live' data elements for each discovered node.

What to Measure?

Taking each of the *Available Nodes* and choosing the most appropriate subset to use for the graphics pipeline presents a number of opposing issues. There are two principal elements - that of configuring the pipeline and that of optimizing performance. This is actually a two-stage subsetting process. The first stage extracts those nodes that provide relevant component capabilities for the chosen pipeline type. The second stage then extracts from these the 'best performing' nodes to provide the optimum available facilities.

As previously discussed, jgViz server nodes advertise multiple possible configurations and, hence, multiple types of configurations. The user is required to specify a few basic requests regarding the pipeline they would like to use. Mandatory are the type of pipeline (either a

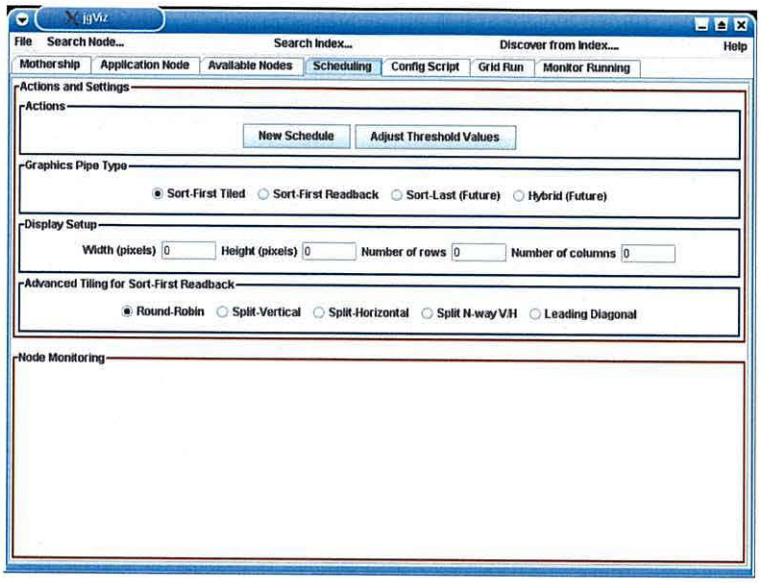


Figure 4.2: jgViz client pipeline selection and description user panel

tiled wall or a readback to a single screen[1]), the number of server nodes required and the output resolution. Figure 4.2 shows the GUI for specifying these.

For a tiled display graphics pipeline, there is only one relevant type of slave node termed *sortfirst-slave*. For readback considerations, the two types *readback-master* and *readback-slave* represent the compositing node and slave render nodes. However, only the *readback-slave* type represents a rendering slave and is used here. The machine to display the composited readback on (the *readback-master*) is chosen by the user from a list of those available when the user elects to schedule the pipeline. The jgViz client matches only the relevant configuration type server resources in the available nodes to produce *subset 1*. Figure 4.3 shows this by means of an example.

In order to optimize performance of the resulting pipeline, it makes sense to try and choose the individual components that are most likely to lead to this. To establish such a collection, jgViz considers two basic data categories - the remote system load as a proportion of its capability and the network latency. To obtain load information, a query is made of the GRIS server on the remote node, thus bypassing any GIIS servers that may have been involved in the initial discovery thereof. The GRIS server running on OpenLDAP allows different cache timeouts to be set for all the data items it 'knows' about. The default setting for

[1]a tiled pipeline being one in which a 'display wall' is used and each tile (screen) is generated by a different machine, and a readback pipeline being one in which seperate tiles are generated by different machines and then readback to a single compositing node for display on a single screen

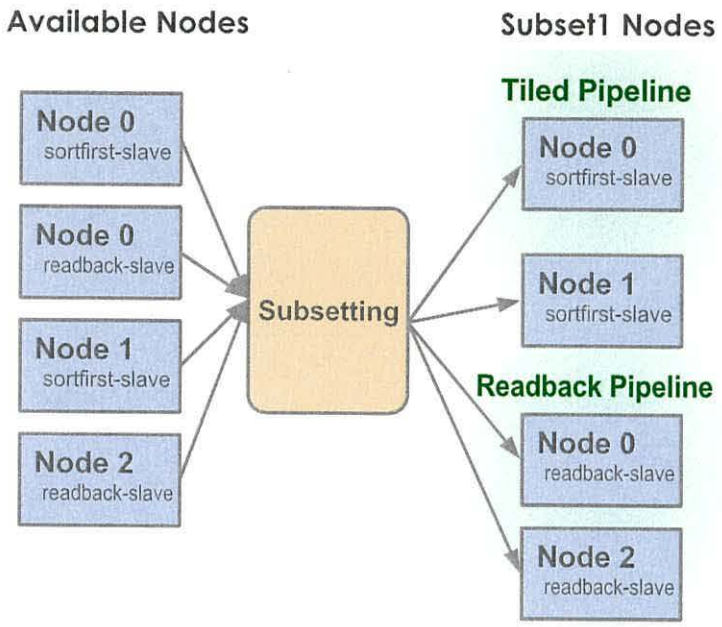


Figure 4.3: Subsetting stage one - by configuration type

time-sensitive data such as load averages is therefore set by default to be low enough so that we can be sure of receiving current data through a refresh of the data we originally discovered.

The other data that needs to be discovered is the status of the network connections involved. Presently, jgViz bases this on latency alone as it is assumed that enough bandwidth exists within the point-to-point links of the network infrastructure being used. jgViz continually and repeatedly launches a small shell script to send *ICMP ECHO* packets to measure the latency. We must use a shell script as Java (our preferred implementation language for jgViz) cannot natively send such ping packets, but it can call a host operating system executable to do so for it. We then take the output from the shell script to obtain values of minimum, mean and maximum latencies and we keep multiple copies of these items in order to calculate consistent averages over time. This approach also avoids placing any load on the target machine and so not interfering and presenting a false view during the process of discovery and scheduling. As we only use the *fork* [2] job launching system to launch jobs immediately for an instant graphics pipeline, it is critical for this data on which

[2] *fork* is the default Globus job manager. A job manager is the remote component that talks the GRAM protocol and handles the job submitted. *fork* simply starts the submitted job immediately, whereas other job managers (such as the Sun Grid Engine manager) submit the jobs into batch queueing systems for handling as-and-when the batch system schedules it.

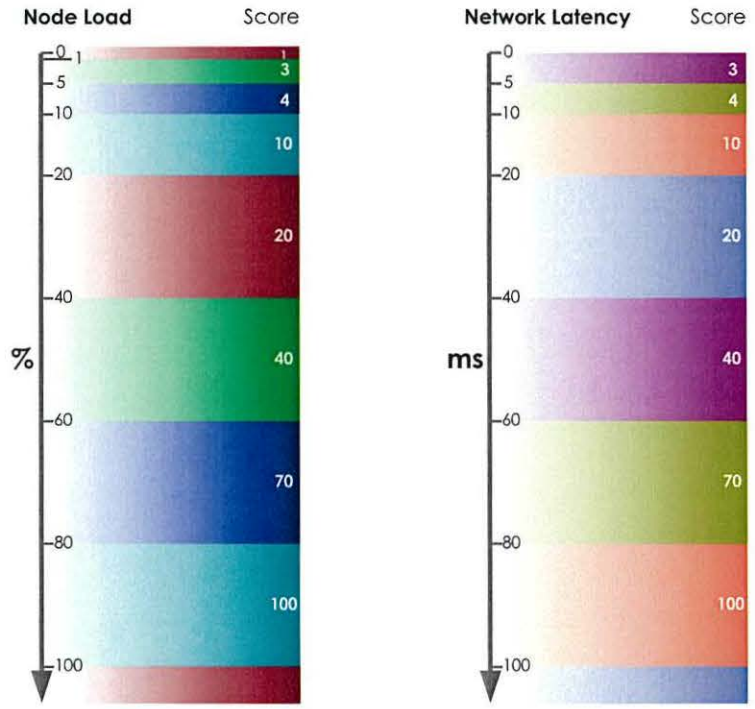


Figure 4.4: Metric result ranges and their default scores for network latency and node load

we base our scheduling decisions to be current.

The two performance metrics are measured and scored independently for each node according to a set of metric-specific result ranges. jgViz contains pre-defined scores for each of these ranges, but they are runtime customizable. The node score is defined as the sum total of the two metric scores. Lower score equals 'better' nodes. Figure 4.4 shows this diagrammatically.

In order to schedule a parallel readback system, jgViz will simply choose the 'best' selection of machines it has available. In order to schedule a tiled display, jgViz will group together and then average the scores of all the nodes involved. The judgement as to what comprises a 'better' solution than another is the same simple scoring system as above. The subsetted machines now become the *Active Nodes*. Once the slaves have been scheduled as necessary, jgViz offers the user the selection of all compositing display (*readback-master*) nodes it found, and the user selects which one they wish to use. This machine is 'tacked' onto the end of the *Active Nodes*. Figure 4.5 shows the second stage of this process.

4.1.2 Configuration Script Building

Once the tiled-groups or individual nodes have each been scored, jgViz simply chooses the required number with lowest score to become *Active Nodes*. In order to actually launch a Chromium session, it is necessary to produce a Chromium configuration script (written in Python) to be executed as the mothership (by using library inheritance to gain and customize mothership abilities). This is responsible for configuring all other Chromium components (all Chromium components contact the mothership immediately when started to establish what they are supposed to do and what other components they are linked to). jgViz generates this configuration script from the discovered resource data (including ordering a tiled display correctly), depending on the chosen graphics pipeline type, to be passed to the runtime component.

In order to make the configuration script the only data-passing method between the discovery/scheduling process and the pipeline launching process, data not relevant to the actual Chromium system but needed for the launch of the pipeline is 'hidden' in the configuration scripts. This is in the form of specifically tagged comments in the Python configuration script. Therefore, Chromium will ignore the comments, but the jgViz client's runtime components can pick out the necessary details. A typical example of such a configuration setting would be the canonical path to the Chromium system on the target machine, as advertised by its jgViz GRIS information provider - Chromium components implicitly know this, but jgViz does not and the location can often vary.

As previously detailed, there are three types of Chromium node - motherships, application nodes and network nodes. The *mothership* takes no active role in the running pipeline and is solely responsible for configuring the components of the pipeline at their startup time via a network connection. There is only ever one mothership for a pipeline. Figure 4.6 shows the configuration script elements for the pipeline, most notably including the application to be executed and the details of Python and Chromium locations on the mothership node.

The *application node* hosts the application to be executed and takes the GL graphics output into the Chromium system. jgViz enables only sort-first pipelines in this first iteration, and so only allows one application node (graphics source) in the pipeline. Chromium contains abilities to do sort-last pipelines with multiple applications (or, at least, multiple input graphics streams), but these are a highly specialised and rare case. jgViz configures application nodes in the same way, irrespective of whether they are involved in a tiled or readback display environment. Besides the application, this configuration also involves

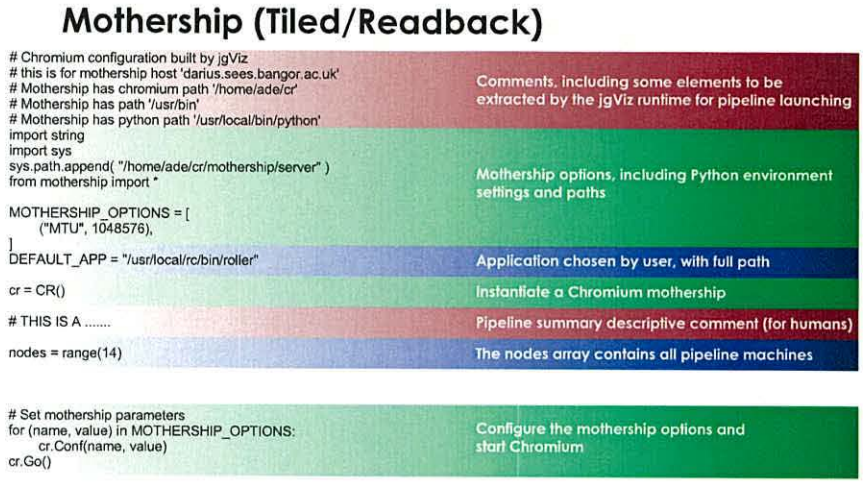


Figure 4.6: Pipeline configuration elements for mothership node

configuring the *tilesort* Stream Processing Unit and the links from it to all render nodes. The operations involved in this process for a live data stream can be computationally intensive, so the application node is typically the most capable machine involved in the pipeline. See figure 4.7 for the script components.

Network nodes form the real backbone of a parallel pipeline as they are responsible for the actual rendering. Such nodes are configured differently depending on whether they are part of a tiled or readback pipeline. In a tiled system, they are responsible for the display of one tile from the entire scene, so they are configured with a 'render' SPU. In a readback system, they are responsible for rendering their tile of the scene and then reading back the pixel data using *glReadPixels()* and dispatching this over the network to the compositing node, which is the last network node defined. This is carried out using two Chromium SPUs. Firstly, the 'readback' SPU renders the tile and performs the glReadPixels() command. Secondly, the 'pack' SPU bundles the resulting GL data together for network transit. So, the definition of a network node is the same in both tiled and readback setups, but the SPUs configured into them are different. Additionally, a readback configuration requires an extra network node to composite and display the image (the *display* node). This need only be configured with a 'render' SPU to accept the GL commands containing the rendered pixels from the network slaves and render them. Where we had to put in a 'pack' SPU previously to send data over the network, a network node has an 'unpack' SPU by default and so does not require manual addition. Figure 4.8 shows the various combinations of components necessary in the configuration script for network nodes and clearly shows the viewport size configured for both 'readback' and 'render' SPUs as well as the path to the OpenGL library (libgl) to

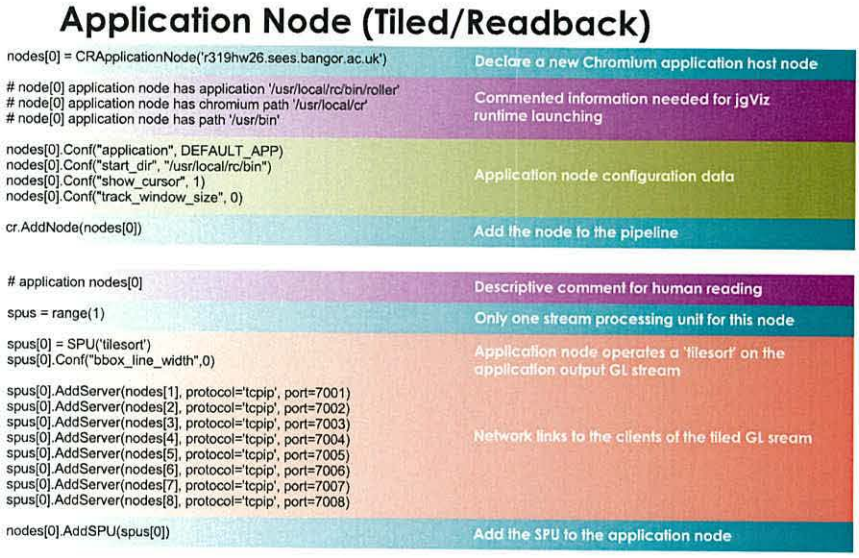


Figure 4.7: Pipeline configuration elements for application node

be used on each machine.

4.2 Conclusions

The optimal scheduling of Grid jobs varies greatly. Graphics jobs specifically can have very difficult requirements depending on the nature of what they intend to do. However, most graphics jobs are likely to have a degree of interactivity associated with them (even if they are not real-time) and so implicitly have tighter timing requirements than non graphics jobs. Additionally, whereas compute jobs can often be submitted to batch queues for execution as and when resources become available, a graphics job will most likely require prompt execution so that the intended 'viewers' can be in the right place at the right time.

The scheduling process within jgViz shares some common functionality with the monitoring process described in chapter 5, in the form of measuring metrics associated with graphics nodes. Scheduling uses these metrics in association with discovered configuration data to develop a pipeline to suit the user's requirements. Depending on the type of pipeline chosen by the user, nodes are either treated as part of a collective group (a tiled display) or individually (for readback purposes). Due to the nature (and in comparison to what we see in chapter 5), the scheduling process cannot adversely affect the machines it 'observes' as they may be busy, so round-robin scheduling is used here to lower the measurement

Network Node (Tiled/Readback)

nodes[4] = CRNetworkNode('r319hw27.sees.bangor.ac.uk')	Declare a new Chromium network node
# network node path '/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin' # network node crpath '/usr/local/cr' # network node glpath '/usr/lib'	Comments containing information for jgViz runtime to extract and use for launching purposes
cr.AddNode(nodes[4])	Add the node to the pipeline

Network Node (Tiled SPUs)

# network nodes[8] = column 3 row 1	Descriptive comment for humans
spus = range(1)	Just one stream processing unit
spus[0] = SPU('render') spus[0].Conf("on_top", 1) spus[0].Conf("window_geometry", [0, 0, 1280, 1024]) spus[0].Conf("borderless", 1) spus[0].Conf("system_gl_path", "/usr/lib")	Define 'render' SPU and rendering viewport size
nodes[8].AddSPU(spus[0]) nodes[8].AddTile(1280, 3072, 1280, 1024)	Add SPU to node and configure tile of the scene view for this node

Network Node (Readback slave SPUs)

# network nodes[12] = readback node	Descriptive comment for humans
spus = range(2)	Two stream processing units for this node
spus[0] = SPU('readback') spus[0].Conf("on_top", 1) spus[0].Conf("window_geometry", [0, 0, 1280, 1024]) spus[0].Conf("borderless", 1) spus[0].Conf("system_gl_path", "/usr/lib") spus[0].Conf("title","cr_readback_r319hw37") spus[0].Conf("extract_alpha", 0) spus[0].Conf("extract_depth", 0) spus[0].Conf("resizable", 0)	Define 'readback' SPU and rendering viewport size
nodes[12].AddSPU(spus[0])	Add SPU to node
spus[1] = SPU('pack') nodes[12].AddSPU(spus[1])	Define and add second SPU – 'pack' - to send glReadPixels readback data to compositing server
spus[1].AddServer(nodes[13]) nodes[12].AddTile(1065, 512, 213, 512)	Add network target and define tile of the scene to be rendered by this node

Network Node (Readback compositor SPUs)

# network nodes[13] = render node	Descriptive comment for humans
spus = range(1)	Just one stream processing unit
spus[0] = SPU('render') spus[0].Conf("on_top", 1) spus[0].Conf("window_geometry", [0,0,1280,1024]) spus[0].Conf("borderless", 1) spus[0].Conf("system_gl_path", "/usr/lib") spus[0].Conf("title", "readback_render_r319hw40") spus[0].Conf("render_to_app_window", 0) spus[0].Conf("resizable", 0)	Define 'render' SPU and rendering viewport size, into which all tiles will be composited
nodes[13].AddSPU(spus[0])	Add SPU to node

Figure 4.8: Pipeline configuration elements for network nodes, showing node configuration for both pipeline types and different SPU configurations for each pipeline type

frequency. We have shown how the scheduling process takes two steps - firstly selecting the resources capable of offering the required configuration from the rendering resources discovered, and secondly selecting the 'best' of those for use. The output of the scheduling process is an augmented configuration script that is passed to the runtime system but can also be exported for separate use.

The scheduling performed within jgViz is a custom process that uses data gained from the LDAP-based, MDS indexing system. This compares with the RAVE project [65] that performs scheduling on the basis of an available server list found in the Universal Description, Discovery and Integration (UDDI) [3] service and the status interrogation of each of the servers found, and the selection amongst available resources discovered from an OGSI-based registry in the RealityGrid system.

In summation, the scheduling process represents the vital link between information gathering and runtime. jgViz's scheduling automates this interface, and optimizes the resulting solution.

[3] `http://www.uddi.org/`.

Chapter 5

The jgViz Grid Runtime

Distributed systems on the scale of Computational Grids present many unique problems and challenges. Alongside the information exchange that takes place within the Grid (as covered in a previous chapter), jgViz makes further direct and extensive use of Grid protocol functionality at runtime. There are several stages involved in the process of taking the output of the scheduling process and turning this into the necessary running Grid components. However, these are best split into two classifications - that of pre-launch and that of post-launch.

Pre-launch is responsible for launching the required graphics pipeline elements on the relevant machines and all tasks necessary to do that, including the establishment of Grid security rights. Post-launch is the smaller task of monitoring the running jgViz graphics pipeline and taking appropriate actions with regard to maintaining optimum performance, for example reacting to changing network conditions. Both of these collections of tasks are necessary in order to provide a dynamic and adaptable distributed graphics pipeline.

5.1 Pre-launch

5.1.1 Network Graphics

Accessing remote graphics resources can be problematic. The X11 protocol [74] is a network-transparent client-server system for the display of graphics on remote machines. This is contrary to the design of systems such as Microsoft Windows, which have not been

designed with remote abilities in mind. The display that is being used runs the X server and applications that utilise such a display are X clients. The X server maps the abilities of the operating system and graphics hardware that it is running on into the X protocol which is accessed by a client application through use of the Xlib library.

Graphics Pipeline Access and Ownership

Whilst the X system does provide great possibilities, it is also a potential security trouble-spot. A number of designs have evolved for providing such security. A brute force mechanism is the simple locking down of the X server to only accept data from certain hosts. Using the *xhost* [75] command, the user can make their local X server accept traffic from other hosts through a simple +/-host syntax and, without arguments, can view current privileged machines. However, this mechanism opens up the X server to all traffic from the remote host which, if it is a multi-user machine, grants full access to the X server to other users and allows them to open windows, kill windows, log key-presses and obtain snapshots of the current display.

A common solution to the above problem is a program called *xauth* [76]. xauth requires a secret string (called a *cookie*) to be known to the client for the server it wishes to talk to. This is kept on a per-user basis, so does not cause opening up of an X server to abuse from any users on a particular client host. Technically, the cookie is called a *MIT-MAGIC-COOKIE*, due to the invention of xauth at the Massachusetts Institute of Technology, and is utilised over IP networks as follows:

On the server:

```
escher:~> xauth list
escher:0  MIT-MAGIC-COOKIE-1  bf9c949db0bb919982ae98b69596b89c
escher-10:0  MIT-MAGIC-COOKIE-1  bf9c949db0bb919982ae98b69596b89c
escher-qfe7:0  MIT-MAGIC-COOKIE-1  bf9c949db0bb919982ae98b69596b89c
escher/unix:0  MIT-MAGIC-COOKIE-1  bf9c949db0bb919982ae98b69596b89c
```

As can be seen, the *MIT-MAGIC-COOKIE* is valid for any of the X server host machine's IP interface addresses (in this case - escher, escher-10 and escher-qfe7 are all a single machine's multiple interfaces onto different subnetworks, implying that the X server is listening on all these interface IP addresses). Also, there is a Unix socket interface on which the cookie is valid. The `:0` represents the display number in standard Unix X server terms (where the

X server for display 0 listens on port 6000, the X server for display 1 listens on port 6001, etc.).

On the client:

```
ariel:~> xauth add escher:0 MIT-MAGIC-COOKIE-1
         bf9c949db0bb919982ae98b69596b89c
ariel:~> setenv DISPLAY escher:0
```

Note that the hostname used (escher) must be a name that can be looked up by the operating system's standard hosts name service (be it NIS, DNS, LDAP, etc.) to get the correct IP for the X server - this may require changing if in a different domain (for example, to being the fully qualified escher.sees.bangor.ac.uk). Secondly, the keys are stored in `$HOME/.Xauthority`, so are automatically propagated in a shared home directories environment - hence, only the `DISPLAY` variable may need to be set. Finally, the X server needs to be started with authority usage, typically in the `xinit` script that is used to startup a user X session (authority is used in all modern Unix operating systems):

```
escher:~> xinit $HOME/.xinitrc -- /usr/bin/X11/X -auth $HOME/.Xauthority
```

Although the above discussion is associated with traditional X usage in a windowed desktop environment, the same set of restrictions apply to production of 3D OpenGL graphics due to the fact that, whether it is software or hardware generated, this produces output on the graphical context held by the X server. The Direct Rendering Initiative (DRI) [77] is a system for safe, efficient, direct access to graphics hardware under the X server. Consisting of changes to the X server, the X libraries and the kernel of the host operating system, DRI was originally conceived for OpenGL rendering. Companies such as nVidia that provide drivers for their own graphics accelerators (under three Unix flavours - Linux, Solaris and FreeBSD) tend not to use the DRI system, however, and instead provide their own modules for accessing the hardware within the standard X.org X server.

Our problem is how to access graphics hardware on a particular remote machine. There are two factors to be considered. Firstly, there may not be an X server running on a remote system [1]. Secondly, the machine may be in use as a standard PC desktop or by somebody else [2], in which case the X server (and thus, the graphics resources) will be tied up and

[1] There is little that can be done to enable an X server in a transparent fashion due to the nature of starting such a service on unknown hardware.

[2] As a workstation or being utilised by other networked - possibly Grid - clients.

inaccessible as discussed above. We currently have no ideal solution to resolve inaccessibility and contention. The research community is actively looking at solutions for this issue, but no overall satisfactory solution has been found that has succeeded in becoming popular. These include off-screen rendering, which has been found in [66] to suffer a (sometimes substantial) performance deficit. In fact there is probably no portable solution that does not involve customization for either jgViz use or desktop use, and in this case the far larger desktop functionality must obviously be the primary target for any single project (such as the X.org project).

Readback Issues

The utilization of remote graphics resources dictates that the rendering process output (i.e. the 2D pixel image) is transported back to the display host. As mentioned in chapter 2 the process of reading back the rendered graphics from the graphics hardware can be slow. In addition to the custom hardware solutions from HP and IBM mentioned previously, there are other considerations necessary.

There are a number of reasons why readback performance can suffer. There are three principal components involved in the rendering and readback process - the graphics processing chip itself, the system interface to it, and the operating system software drivers for it - all of which are relatively dynamic. Graphics processor lifecycles are 12 months between major revisions and 6 months for minor revisions (clock bumps). Software drivers are regularly revised, in particular nVidia provide very regular updates, optimizations and fixes for a number of operating systems (currently Windows, Linux, FreeBSD and Solaris [78]). Only the system interface for the graphics processor is less frequently updated, although updates can still be expected every two or so years.

Since 1996[3], the focus with graphics processing chips has been on pure rendering performance to the local screen. This is an approach largely driven by the home consumer market. During the development of such hardware, readback performance has often suffered as it is not a requirement in such a market. Nevertheless, the last year or two (alongside improved system interfaces) has seen significantly improved performance readback performance. A significant part of this has been the changeover from the Accelerated Graphics Port (AGP) system interface to the PCI-Express system. AGP's offering of bandwidth up to 2133

[3]1996 was the year in which 3Dfx released the Voodoo PC graphics accelerator, which was the first affordable, high-performance 3D card for the PC. The so-called 'killer app' for Voodoo was the OpenGL subset driver (called the MiniGL driver) developed to run the id Software came Quake, and thus kicking off the 3D gaming age

MB/Sec to the graphics chip in the AGP8X form is outdone by PCI-Express's bi-directional 4000 MB/Sec in its standard graphics form of 16-channel PCI-Express.

Additionally, due to the degree of programmability in place in modern graphics chips (which is being increasingly exploited in other ways [79] [80]) and the crucial interaction with the operating system and its use of non-graphics hardware resources, the software driver for graphics hardware is of considerable importance. In line with the generally increasing awareness of readback, software drivers have moved toward providing improved performance in the area, and large performance differences are sometimes seen between different driver versions.

Traditionally, both ATI and nVidia graphics processors have suffered from performance hits in performing readback function, although it has been subject to variation with different driver versions. However, that has now changed somewhat as recent nVidia chips in particular using PCI-Express system interfaces have seen improved readback performance. The present to future timeframe allows us to envisage systems consisting of new, increasingly programmable graphics processors supplied with data over a non-direction preferential system interface by an operating system's graphics subsystem. Through increasingly optimized software drivers, they will know how to communicate with the hardware for maximized performance.

5.1.2 Network Protocols

jgViz's distributed graphics pipelines are heavily dependent on the networks that connect the various components. As well as the capability of the hardware in these networks, related software layers are also equally important. The nature of the Grid dictates that we desire both high performance and high security. It is enticing but impossible to produce a single protocol that can handle all varieties of usage, however, there have been many different Grid data transport protocols produced. Globus XIO provides a standardized API for plugging in different Grid transport protocols. It is based on byte-transfer schemes and hides as much detail as possible from the user, but also is designed with the knowledge that some tuning of protocols will be necessary to obtain maximum performance. GridFTP is the most popular system for transferring files of data among Grid resources and it offers advanced features such as striping and the use of replica servers. However, as an extension on the FTP protocol, it is not intended for data streaming.

Due to the nature of visualization data streaming, we cannot make use of performance

enhancing facilities such as single-file parallelism and striping. As the stream is 'live' data, multiple-source transfer striping is impossible. However, future work may investigate in more depth the possibility of SMP machines performing parallel data transfer - this may be particularly helpful at the bottleneck points of pipelines. Our requirement for maximum performance also reduces the amount of processing overhead that is acceptable. The RealityGrid project utilised the sockets-based Globus I/O subsystem to provide the necessary performance [62] as SOAP over HTTP cannot provide the necessary performance. Kola, Kosar and Livny [81] carry out a profiling of Grid data transfer systems to study the parameters of transfer rate, parallelism and system load using GridFTP and the NeST [4] transport protocols. The resulting data suggests fairly high CPU utilization of relatively powerful machines in parallel setups. This is particularly relevant to the jgViz components that are responsible for distributing and collecting together data. The fact that it is found to be so necessary to 'tune' the balance of parallelism and protocol parameters suggests that such systems would limit jgViz performance. Similar performance data is seen in [82]. Hence, we have chosen, for the time being, to use Chromium's native transport in order to minimize processing overhead and maximize pipeline performance. As further work goes on in the area of real-time data streaming, Grid native transport protocols are very interesting, particularly as we further study wide-area use of jgViz, rather than the localized setups we are currently concerned with. The possibilities of making use of network intelligence, such as multicasting for multiple-viewers and Quality-of-Service systems for guaranteed bandwidth, promise much for jgViz as they have already delivered in other projects [5].

5.1.3 Launching Components

Once the jgViz client has discovered available Grid resources and scheduled an appropriate graphics pipeline, it is only necessary to launch the relevant Chromium components on the selected machines. This process takes place in two parts - establishing the machines and settings to be used, and then utilising them to launch the pipeline.

Job launching in Grid environments is a many-and-varied thing due to the diversity and number of submission and queueing systems. The requirements of any system involving interactivity restrict substantially the options for the scheduling-execution system that can be used - for example, utilisation of a batch control system would result in substantial coordination problems amongst multiple machines in addition to the loss of an 'on-demand'

[4] Network Storage, see `http://www.cs.wisc.edu/condor/nest/`.

[5] Such as the AccessGrid media-conferencing system.

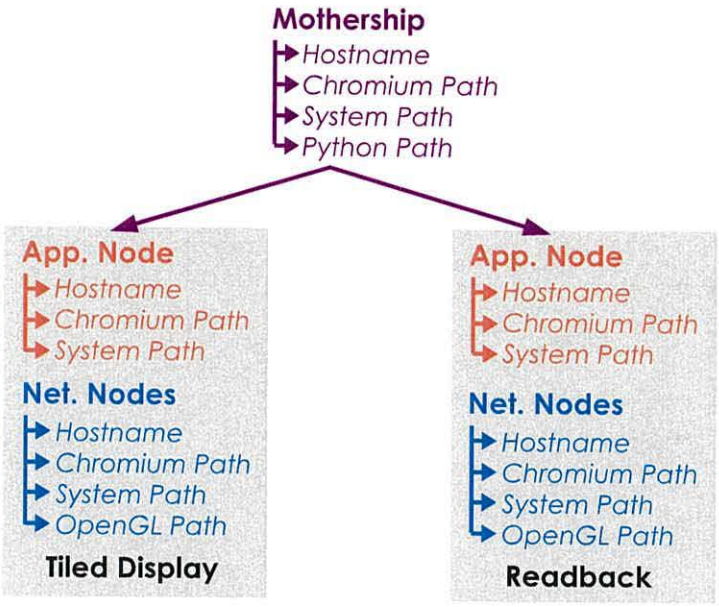


Figure 5.1: Data items for each node type within the Chromium configuration script

service. Obviously, it is a critical part of the envisaged jgViz visualization system that the system does make available a near-instantaneous service, within the bounds of available hardware.

Configuration Parsing

The configuration script output from the scheduling process is editable for customization purposes by the user. However, a final jgViz system requires that this ability is removed in order to protect available Grid resources from being utilised in ways not desired by their 'owners'. For the purposes of this research, however, allowing this has made certain testing and development easier.

With all the necessary information available in the configuration script that the jgViz client runtime accesses, the data is extracted through simple linear searching for the known 'tags'. Figure 5.1 highlights that which is extracted in order to launch the relevant Chromium components using the Grid protocols.

Grid Execution

As previously highlighted, security is a key design element of the Grid. Several component parts comprise this security solution. The concepts of authentication and authorisation are key, whilst encryption can also be used depending on the exact nature of the requirements. It is a fundamental part of executing on remote grid resources that the user is authenticated to ensure that they are who they say they are. It is also necessary to authorise the user to access the requested resources within established policy parameters. As a Grid application, jgViz utilises the established protocols for performing job launch over the Grid. This in turn requires that a Grid 'proxy' is set up. A proxy is a finite-lifetime set of credentials that authenticate a user through their certificate and passphrase. For the lifetime of the proxy, Grid protocols can use the credentials to access Grid resources. Presently, jgViz expects the user to have a proxy setup when it launches, and will give an error if this is not the case. This solution was chosen due to incompatibilities observed in the way that proxies are created and credentials are stored, using the Java CoG Toolkit compared to the Globus Toolkit. Future work would embed this creation capability within jgViz. A proxy can be setup using the command-line utility that comes with Globus or a graphical one that is part of the Commodity Grid Toolkit.

Prior to launching the actual pipeline components on the Grid, jgViz tries its best to verify Grid connectivity. This is achieved by sending a GRAM (Globus Resource Allocation Manager) 'ping' to each machine involved. This is a Globus runtime layer dummy function that returns no result and is simply used to test that a Globus gatekeeper is running and accepting jobs. The fact that a GRIS index server is running and so advertising a Grid resource is not a guarantee that it is reachable as they are different daemons listening on different ports (in fact, the Globus Runtime port is usually monitored by the *inetd* super-server that instantiates Globus handlers dynamically as required). If any of the node pings fail, then jgViz will not try to launch any of the pipeline.

Once the pre-requisites above are concluded, jgViz can move on with the actual process of launching the graphics pipeline components. This is a multi-stage process in itself. As the parsing of the configuration script has taken place, jgViz now has all the knowledge needed. Figure 5.2 presents a diagrammatic overview of the launch process, as discussed below.

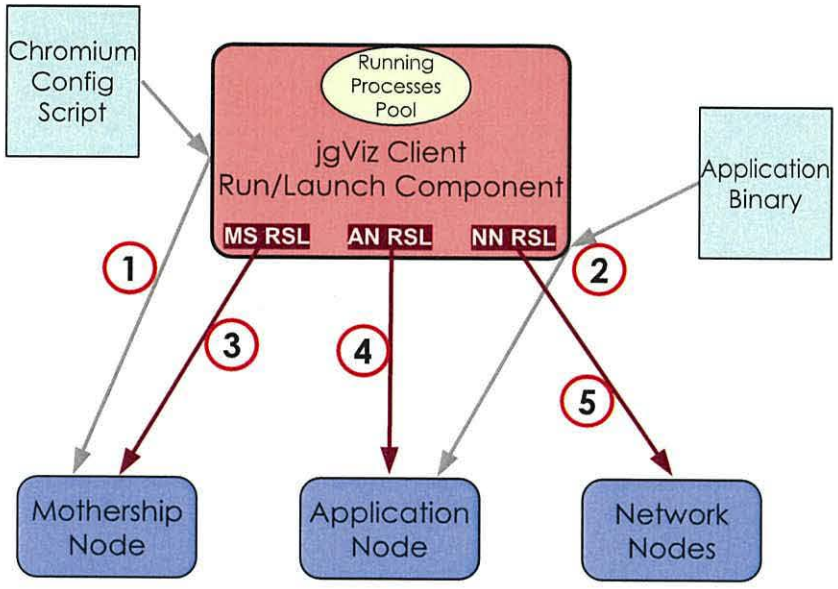


Figure 5.2: Components and Ordering Involved in Launching a jgViz Session

1. Transfer of Chromium configuration script via secure GridFTP protocol.
2. Transfer of client-side held application binary via GridFTP.
3. Generation of RSL (Resource Specification Language) and launch of the mothership using Python and Chromium path data, and a transferred configuration script. Followed by a slight pause to ensure that the mothership is up and running and accepting connections.
4. Generation of RSL and launch of application node using Chromium path data and a remote application binary.
5. Generation of RSL and launch of network nodes using Chromium and *libgl* path data.

GridFTP Transfer

The GridFTP protocol is utilized during the jgViz runtime processes for the transfer of a number of important files. During the pre-launch procedure, the transfer of the mothership configuration file (as previously generated within jgViz) to the Chromium mothership node is critical. This must successfully take place so that jgViz can then launch the mothership component via the Globus runtime and a call to the *Python* interpreter on the remote machine. Pakhira *et al.* [83] question the compatibility of GridFTP and firewalls when dealing with large data transfers and go on to utilise the Storage Resource Broker (SRB)

Grid service instead. We are not presently operating in a firewalled environment and have seen no problems, and one would also expect that such a problem is due to configuration issue, as the theory regarding such use is valid. It is also speculated that this could be due to their use of the Geodise Computational Toolbox.

Additionally, the user selects whether the application binary concerned is located on their selected application node or located locally where the jgViz client is running. If local, then jgViz will also transfer the binary to the Chromium application node. GridFTP itself is a Grid standard protocol that uses Grid security credentials to authenticate users and provides secure transfer of data as well as partial file transfer and a number of techniques (parallel streams, buffer size customization, multiple-server striped transfer) to improve performance. The Commodity Grid Toolkit implements a simple *GridFTPClient* class that we utilise within jgViz. The server-side process is built and installed as part of the Globus Toolkit and started from *inetd* (inetd is a daemon that listens for connections on specific ports and starts appropriate programs to handle the services on these ports when connections are received).

RSL

The Resource Specification Language (RSL), used to specify Grid job characteristics, is key in launching Chromium components in a non-interactive environment, and we need to make use of several parts of the RSL syntax for this. The relevant parts all represent a component of how to launch an executable in a Unix shell. These parts of the RSL specify the environment, the executable, the arguments to the executable, and the handling of output and error streams:

(environment=(*variable1*)(*variable2*))
Sets up environment variables as necessary. For each of the Chromium components, we need to use a subset of:

1. **PATH** - Sets the search directories that will be searched for executables.
2. **LD_LIBRARY_PATH** - Lists the directories that will be searched by the run-time linker (ld) for library functions.
3. **CRMOTHERSHIP** - Tells non-mothership Chromium components where the mothership is, such that they can discover at runtime the details of their intended functionality.

4. **CR_SYSTEM_GL_PATH** - Specifies the path to the system OpenGL library (*libgl*) that should be used by Chromium. This may be necessary as Chromium requires a thread-safe GL library that is sometimes not the default (in */usr/lib*) for an OS. A typical example is the nVidia Linux drivers, in which the thread-safe libgl is located in */usr/lib/tls*.

5. **DISPLAY** - To ensure that the local X server is used i.e. the local graphics accelerator.

(directory=*path***)**
Working directory for the process to be launched.

(executable=*executable path***)**
Specifies the binary executable to be launched, which will vary depending on which Chromium component is being launched.

(arguments=*arguments***)**
Arguments to the executable, which may or may not be needed for Chromium components.

(stdout=*stdout path***)(stderr=***stderr path***)**
Handling of the stdout and stderr streams for the executable. As Unix is based on the idea of representing everything as files, which can in turn be stored on secondary storage, we can utilise the Global Access to Secondary Storage (GASS) protocol here. Using the Java COG toolkit, we create a GASS server on the jgViz client machine and use the HTTPS protocol to transfer the two streams from the remote Grid machine back. These are then handled as streams by the jgViz client.

Table 5.1 shows the combination of these RSL components used for each type of Chromium node.

RSL Component	Mothership Value	Application Node Value	Network Node Value
Environment Variables			
$PATH	Cr Mothership Path	Cr Binaries Path	Cr Binaries Path
$LD_LIBRARY_PATH	Cr Libraries Path	Cr Libraries Path	Cr Libraries Path
$CRMOTHERSHIP	-	Mothership Name	Mothership Name
$CR_SYSTEM_GL_PATH	-	-	Custom Config
$DISPLAY	-	-	*localhost:0.0*
Executable Arguments	*python* *tmp/jgvmothership.py*	*crappfaker* -	*crserver* -
I/O Streams	STDIO and STDERR via HTTPS to GASS server	STDIO and STDERR via HTTPS to GASS server	STDIO and STDERR via HTTPS to GASS server
Directory	*/tmp*	Cr Binaries Path	Cr Binaries Path

Table 5.1: RSL components used for different jgViz nodes

A typical jgViz RSL statement is, therefore, as follows (formatting inserted for readability):

```
{&(environment=
    (PATH /bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/home/ade/cr/bin/Linux)
    (DISPLAY localhost:0.0)
    (CRMOTHERSHIP darius.sees.bangor.ac.uk)
    (CR_SYSTEM_GL_PATH /usr/lib/tls)
    (LD_LIBRARY_PATH /home/ade/cr/lib/Linux))
  (directory=/tmp)
  (executable=/home/ade/cr/bin/Linux/crserver)
  (stdout=https://147.143.9.26:37169/dev/stdout)
  (stderr=https://147.143.9.26:37169/dev/stderr)
```

We express the runtime environment of the various Grid-launched components as completely as possible, in order to avoid possible problems relating to library search paths and runtime directory locations [83].

GASS

The combination of Unix's modelling of everything as a file (or at least, as an identifiable part of the file system) and GASS is a powerful one. Global Access to Secondary Storage allows any of the Unix 'files' to be handled in some manner via a Grid infrastructure. As mentioned above, jgViz makes use of this facility to handle the standard out (stdout) and standard error (stderr) streams from the various Unix processes it launches remotely, as involved in the Chromium pipeline. The Java Commodity Grid Toolkit contains a class for establishing a GASS server on a port using HTTPS as the transport protocol. The launched processes are set up to use this with the *stdout* and *stderr* RSL substrings shown above. Although jgViz does not currently do any, it is possible to monitor these output streams (particularly the stderr stream) to determine when problems occur. Presently, jgViz simply displays the output of the two streams during the launch process on the 'Run' tab of the GUI.

The Running Pipe

Once the nodes in all three classes of node that are required for a Chromium pipeline are launched, they communicate with each other to establish their settings and begin to operate. As the mothership is the node responsible for holding the configuration for the other Chromium components, non-mothership nodes do not require any configuration other than being told on what machine the mothership is running. This is set using the

CRMOTHERSHIP environment variable as shown above. Once all components are up-and-running, the pipeline starts operating - the application begins and the output is consumed into the Chromium streams system. Each running component is also intimately linked with a Globus Runtime job that provides various capabilities to the program that launched it (i.e. the jgViz client). In order to maintain control over the pipeline, the jgViz client maintains (in a Java vector) all the Globus jobs. Additionally, vectors of the *stdin* and *stderr* handling GASS servers are kept.

5.2 Post-launch

Once a graphics pipeline is up and running, then jgViz takes on the role of controlling and monitoring. The diversity and nature of Grid environments defines that resources stand to become unavailable or of limited availability at any moment. For example, the Internet as a whole is a large and diverse interconnection of hardware and administrative domains in which connections of various capabilities are frequently coming online and going offline. Additionally, the fact that the Grid encompasses not just a specialist research network, but a collection of networks that are used by all users' traffic, means that traffic levels will vary wildly. Alongside the increasing level of intelligence going into the networks we use, with features such as multicast and Quality-of-Service (QoS) protocols, all this means that the network is not predictable or consistent. The Grid protocols and Grid software must therefore, by definition, be built with failure-tolerance in mind. The hierarchies and duplication in evidence with protocols such as the Meta Directory Service (MDS, as covered previously) are good examples of this. In terms of compute facilities, the same is true once again. They may be open to a number of people for use for a variety of differing applications, of which we must assume that none is considerate and gives-way to any other. Therefore, whilst it is reasonable to expect that some form of least-load scheduling would be carried out by any Grid application, we cannot guarantee it and it is necessary for any performance-critical application to guarantee that its own performance is maintained at the necessary level on the hardware in use. Real-time visualization, such as that of jgViz, is a good example of this and jgViz therefore needs to closely watch the running pipeline and take action to ensure it is performing optimally.

5.2.1 Session Control

Before considering the functionality necessary to monitor and deal with a running pipeline, jgViz must be able to control the pipeline. This involves the ability to stop the pipeline and reset the system, thus re-verifying Grid connectivity and allowing a new pipeline instance to be launched.

Stopping a Pipeline

Stopping a running pipeline is not just a case of terminating the Globus Runtime jobs (via the unique job identifiers returned upon each job launch) - to do so would result in an orphan process being left in an undetermined state and without the ability to monitor or terminate them via available Grid protocols. To terminate a pipeline, we need to stop the Chromium mothership and the running application. In both cases, we use a further GRAM job submission to achieve this. Chromium provides a number of binaries for acting on a running mothership - here, we are interested in the *quitms* binary for shutting down a running mothership. For closing down the remaining parts of the pipeline, we only need to quit the application and this will have the knock-on effect of closing the sockets and pipes that pass the graphics data through the separated Chromium components. In order to kill the application, we issue a GRAM job using the Unix *pkill* command. *pkill* is designed for signalling processes - it then depends on the process' signal-handler as to what happens. jgViz includes the capability to send a number of Unix *signals* to the application. For reasons outlined in the next section, the process of stopping an application node therefore consists of first sending a *SIGHUP* (hang-up) to the process in order to signal the suitable signal-handler coded into the Chromium component to end, before sending the *SIGQUIT* to actually tell the process to end. Typical RSL syntax generated and submitted by jgViz for the ending of the mothership and application node (with the *roller* application) is therefore as follows:

```
{&(environment=PATH /usr/bin)
    (LD_LIBRARY_PATH /home/ade/cr/lib/Linux))
  (directory=/tmp)
  (executable=/home/ade/cr/bin/Linux/quitms)
  (stdout=https://147.143.104.90:32867/dev/stdout)
  (stderr=https://147.143.104.90:32867/dev/stderr)}


{&(environment=(PATH /usr/bin)
    (LD_LIBRARY_PATH /usr/local/cr/lib/Linux))
```

```
  (directory=/tmp)
  (executable=/usr/bin/pkill)
  (arguments=-HUP roller)
  (stdout=https://147.143.104.90:32869/dev/stdout)
  (stderr=https://147.143.104.90:32869/dev/stderr)}


{&(environment=(PATH /usr/bin)
    (LD_LIBRARY_PATH /usr/local/cr/lib/Linux))
  (directory=/tmp)
  (executable=/usr/bin/pkill)
  (arguments=roller)
  (stdout=https://147.143.104.90:32871/dev/stdout)
  (stderr=https://147.143.104.90:32871/dev/stderr)}
```

Once a pipeline has been stopped, the jgViz client needs to clear out all reference to the relevant components. As discussed in Appendix B, it is necessary that the client maintains a number of references and so it now 'resets itself' by dropping all such references.

5.2.2 Monitoring Active Sessions, Dynamic Scheduling and State Transfer

Monitoring of a running pipeline is concerned with changing network and machine conditions. jgViz therefore repeatedly measures the network latency and load levels of running nodes, as discussed below. jgViz establishes consistent values for these, and if they increase beyond a user-definable (but sensibly preset) proportion, then jgViz will react.

This reaction takes one of two forms depending on the application being run and is performed as part of the process of stopping the existing pipeline. If the application is custom-built and keeps state related information in a temporary "check-point" file, then jgViz can be told what this file is and 'rescue' it from where it is running via GridFTP before shutting down the pipeline (as shown in Figure 5.3). If the application does not do this and is just a stateless code with no state data (for example, a model viewer), then the pipeline will just be shutdown.

jgViz then returns to the list of available jgViz servers previously discovered and obtains updated metric data for these nodes. The time taken to do this needs to be short enough not to interfere with the perceived quick movement of the pipeline, but long enough to acquire updated metric data on available nodes. On balance, we have chosen 5 seconds for this. jgViz then performs a new 'scheduling' to produce a refreshed subset of active nodes,

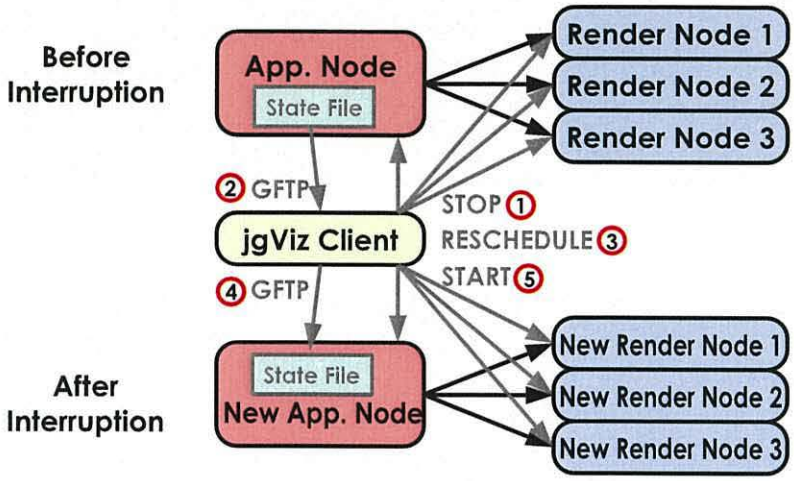


Figure 5.3: The jgViz client reschedules a pipeline

as covered in chapter 4. The scheduling is performed with the same options as chosen by the user for the initial scheduling, but with the updated performance metric data. jgViz will then automatically create a new Chromium configuration and transfer it to the new mothership host for launching.

If a "check-point" state file was rescued it is restored to the new application host via GridFTP and the new pipeline is started with the application picking up the state file and continuing where it left off. In the case without a recent state-file, the new pipeline is just started with the application at its initial start point. This is not an ideal solution of how to preserve the state of an application between pipeline instances as it requires a compliant application, however other options such as taking an imprint of memory content within a Grid environment were deemed far too complex and problematic for the timescale of this project.

Measuring Running Nodes

jgViz performs monitoring of a collection of jgViz servers at two points - firstly when deciding which machines to use in a pipeline during scheduling, and secondly when monitoring a running pipeline. Whilst the scheduling-stage monitoring is intended to be non-invasive on machines that may already be executing jobs for other users, the runtime-stage monitoring occurs at a time when the user 'owns' the machines that are part of the pipeline. Both stages could potentially be involved in monitoring a significant number of machines. We also

		Round-Robin			Concurrent	
Number of Nodes	Cycle Time (ms)	CPU User Time (%)	Memory Utilization (MB)	Cycle Time (ms)	CPU User Time (%)	Memory Utilization (MB)
1	1232	1.1	80	1192	1.1	80
2	2446	1.1	81	1275	2.2	82
4	4917	1.1	83	1439	2.8	90
8	9366	1.1	87	1554	3.4	96
16	18753	1.1	89	2137	4.4	97
36	42038	1.1	90	3865	4.9	98

Table 5.2: Performance results for round-robin vs concurrent monitoring scalability tests

have to consider the performance deficit introduced upon the jgViz client by the monitoring load demanded by the configuration. Unless we look at increasingly complex systems, there are two basic schemes that can be used for monitoring: *round-robin* and *concurrent*. In order to judge which of these systems is appropriate for the two stages of monitoring, a non-jgViz client test was performed. Of the two metrics, the performance of the ICMP PING in hardware requirement is insignificant, even when scaled to many more machines than we are concerned with. However, the machine load metric, as gained from the Grid MDS LDAP server running on the monitored nodes, is not predictable or as lightweight. Table 5.2 and Figure 5.4 show the results of a simple looping test with the two types of monitoring.

These results were obtained using a Sun Blade 2000 with dual 1.05GHz UltraSPARC-III processors and 8GB memory for the jgViz client. Whilst the concurrent monitoring implementation shows vastly improved scalability (in terms of cycle time), it also consumes substantially more client-side resources. When concurrently monitoring more than 8 nodes, the limiting factors become I/O delays and the additional processing associated with the LDAP search results. Of course, having a number of threads trying to execute on the same processor in parallel also produces a contention issue, reflected in the fact that the system-load (which is a measure of the average runqueue length) being subject to a greater proportional increase than the pure CPU user time. This also explains why the machine appeared to be a lot slower to use, although the CPU utilization does not indicate as such.

Monitoring of the pipeline that takes place once it is up-and-running, this could be potentially detrimental if, for example, the application is hosted on the same machine as the

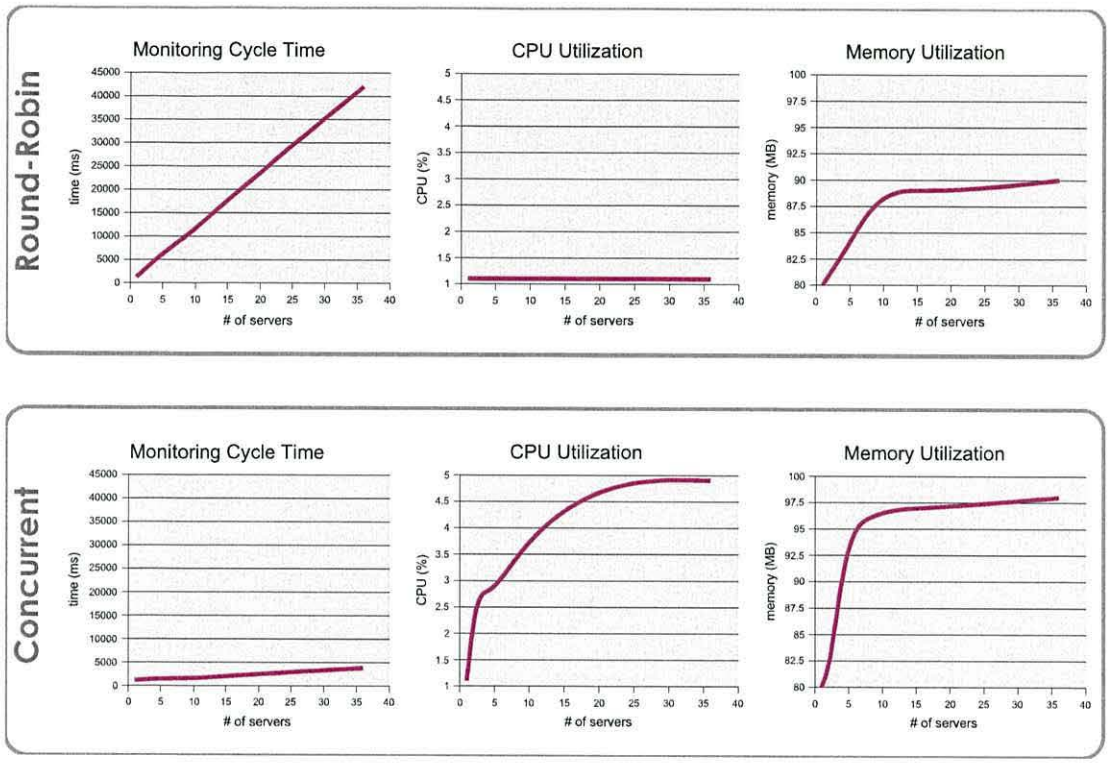


Figure 5.4: Graphical representation of round-robin vs concurrent test data

client. However, the additional network and server load caused by this is unlikely to be significant as the checks are still taking place at least one second apart for any single node and are lightweight. As the number of monitored servers increases, then the jgViz client has more work to do and so the question is posed of how many servers a typical institution is likely to have that would need to be monitored. As a result of this, scalability across systems of increased distribution and node-count will require further thought. However, within jgViz's current target scale, it is a fair conclusion that the round-robin monitoring is well-suited to use at the non-running, scheduling stage but that it would provide insufficient performance during the runtime stage. Therefore, the concurrent system is the logical choice for use here. Note, however, that additional limitations are placed on what other tasks the jgViz client machine can do and on the class of machine one might expect to use for such a task in the first place.

Triggering an Event

There are a number of different, competing requirements in monitoring a running jgViz pipeline and deciding that it is suffering enough in performance that it should be stopped and moved to a different set of machines. The overall goal is obviously to maintain and enhance the pipeline performance. Equally, it is desirable that the impact of monitoring the pipeline is kept low on the pipeline component machines. We also wish to be prompt in spotting and moving a pipeline, so we need to be able to make quick decisions. However, the relocation should not be too hasty and move a pipeline that is running well, so we must be precise in judging that which needs moving and that which doesn't. This, in turn, works against a prompt response, as the sometimes 'peaky' data that can be seen for the two metrics involved (also a consideration with a visualization that does not run at a consistent, relatively smooth frame rate) does not lend itself well to decisions based on very short duration performance data.

These requirements and limitations are difficult to balance and could, on their own, produce a work of immense study and depth. Within jgViz, we try to take a relatively simple top level approach and allow easy 'fine-tuning' of parameters within that. The approach taken is to keep two moving historical sets of data and compare the average data values in them. That is, the average of a long-term history, the average of a short-term history and the proportional difference between them. When the short-term average is more than the set proportion greater than the long-term average, jgViz triggers an 'event' that represents pipeline performance having dropped to a level requiring attention. The stop, reschedule, start process then begins as detailed.

When in monitoring mode, jgViz repeatedly loops through all nodes updating load and network latency data. On each cycle, this data is added to the short and long term history arrays for each metric and the new average of all the data in each is calculated for comparison. jgViz uses six customizable parameters to dictate the outcome of this comparison. The six parameters are:

1. Network latency long history buffer size.
2. Network latency short history buffer size.
3. Node load long history buffer size.
4. Node load short history buffer size.

5. Required percentage increase in network latency short history buffer average compared to network latency long history buffer average.

6. Required percentage increase in node load short history buffer average compared to node load long history buffer average.

Given the performance levels concerned in concurrent node monitoring (with short cycle times of the order of one second), the level of definition in different operating system's 'load' metrics and the rare but occasionally significant variations in millisecond-order network latencies, deciding on the right combination of these settings can be very difficult.

Figure 5.5 presents a diagrammatic example of the monitoring-event process. It can be clearly seen that the network latency associated with node 1 has increased by a suitable proportion to cause a trigger, which will cause jgViz to stop the pipeline and reschedule it.

Monitoring Other Metrics

The existing two metrics that jgViz uses (network latency and node load) are deliberately chosen for their 'big-picture' qualities of being relatively light to test and also very indicative. In compute resource utilization terms, neither individually places a restrictive load on either the client in checking them, or the server answering such check requests - in which, an LDAP query is the most intense operation required and so which would only be expensive if the machine were particularly busy on something else that had to be interrupted, either in terms of CPU time or virtual memory paging. In network terms, an ICMP PING is very light in load, principally because the majority of network cards available now handle such echo requests with minimal processor interaction. The two existing metrics' light-weight nature allows them both to be used during the initial scheduling process and the runtime monitoring stage.

There are a number of other general considerations relating to the use of different metrics. For distributed systems, bandwidth costs money. For all modern computer systems, power and cooling cost money - and increasingly so all the time. However, these costs may be insignificant next to the costs of people. Employing administration personnel with appropriate knowledge to configure and operate Grid facilities will likely be a significant cost.

The possibility of using other metrics beyond the base pair is an enticing one, in order

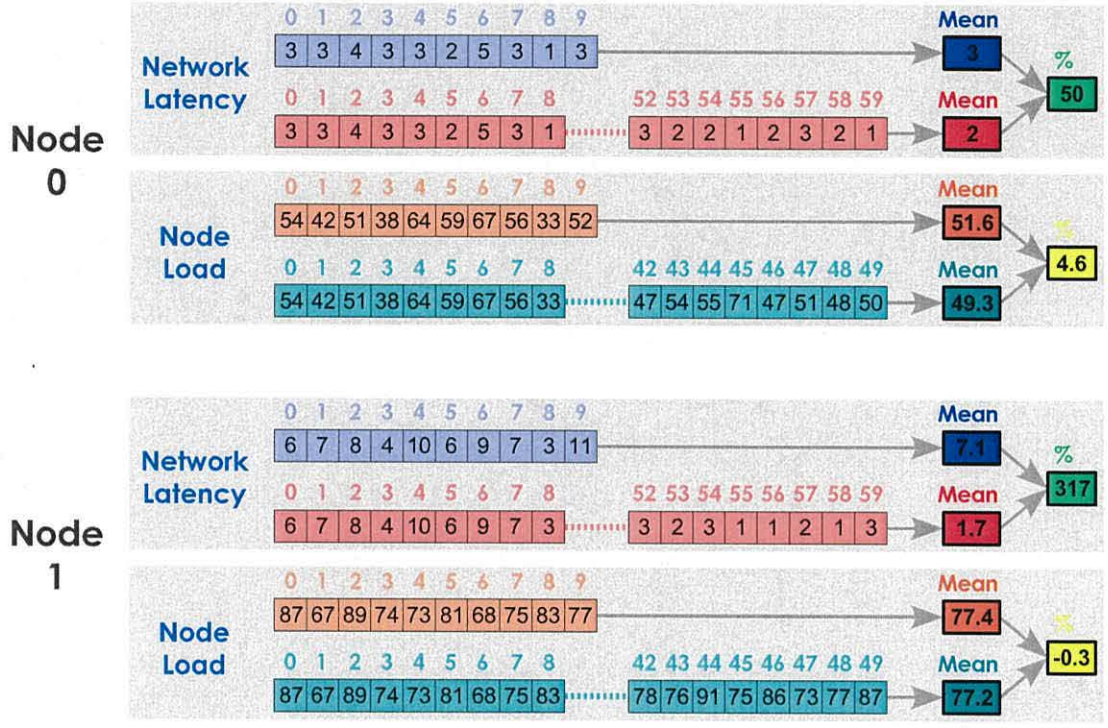


Figure 5.5: jgViz client's monitoring process, showing two historical buffers (short and long) being kept for each of two metrics (network latency and node load) along with the associated calculations and parameters, for each of two monitored nodes.

to improve the quality of the jgViz experience. Additional metrics will impact upon this and the above mentioned factors and all must be considered. There are three areas that additional metrics of various kinds can be considered - node hardware capability, network state and application classification.

It is exceptionally difficult to assess the real-world performance of a system. There is such large variation in chipsets, processors, memory, disks, operating systems and many other components - all components that work below the jgViz and Grid layers but which vary in terms of quality, throughput and outright 'speed' and, thus, have a knock-on effect. For the purposes of our network (render) nodes, we are focused on the rendering system requirements. However, that still involves a significant number of components and is thus subject to large variation. Statistical metrics, such as geometry performance (polygons per second) or texture performance (pixels per second) can be established for individual graphics accelerators, but are subject to the rest of the hardware in a system. Likewise, evaluating CPU and memory effects presents similar problems. Particularly as the interfaces between parts of computer systems evolve, components will be increasingly subject to factors external to themselves. For example, the industry has recently seen the introduction of dual-core CPUs and new CPU-system interfaces in which placement of items such as memory controllers has changed (AMD K8 series processor systems using the HyperTransport front-side bus). Within the graphics rendering field, however, the interface to the graphics accelerator is important and just as difficult to measure the performance of. 3D rendering benchmarks, such as the 3DMark series, may provide a solution. However, they are typically provisioned only for the Windows OS, creating problems in a heterogeneous environment, and with a relatively specialised field of 'expertise' (games). Gaining such a performance evaluation for a machine would likely be very useful for the initial scheduling carried out by jgViz. However, it also requires additional setup time from an administrator. For the purposes of runtime monitoring (alongside initial scheduling when nodes are already being used for other purposes), it is difficult to see what other metrics could be utilised without adversely affecting the running system. Additional metrics on virtual memory subsystems and CPU statistics are available, but vary amongst different operating systems and amongst different versions thereof, and are difficult to make use of without thought to advertising quiescent levels of the same statistics for comparison. Even then, the amount of 'artform' involved in operating system low-level features such as schedulers and virtual memory systems may restrict the usefulness of such data. The possibility of running small test programs to test various real-world capabilities of such hardware presents similar problems in the required full-occupancy of machines that may already be in use or, in the case of being pre-evaluated, work required from the system administrator. Again, these problems are accentuated in a

heterogeneous Grid environment.

Networks present similarly themed problems as rendering nodes. The principal concern is the technical one of how difficult it is to evaluate the performance of a network at a particular time without adversely affecting other users of that network. The financial implication may be additional to this in some Grid-utilizing establishments. Once again, it is possible that the systems administrator could do some additional setup involving real-world performance measurement, and that such data could then be advertised by the jgViz server-side GIS backend. Another possibility is to advertise the theoretical performance available depending on the networking technology in use. This then has the advantage that it should be possible to setup without system administrator time, but obviously is not a practical measurement. Issues such as cable quality, cable length, network switching capacity and so-on all have an effect on this. The Network Weather Service (NWS) [84] offers a general-purpose solution here that has recently been implemented within the Globus middleware. NWS focuses on enabling distributed scheduling to take place over wide geographical scales of networks in a ubiquitous and non-intrusive manner. Additionally, NWS provides accurate prediction of future network traffic levels, which could be utilised withing jgViz alongside some estimation of the time required for pipeline operation to provide an optimized initial pipeline scheduling. NWS and projects such as the UK's GridMon fit in with moves by the Global Grid Forum to provide a standardized, service-based approach to network performance monitoring and data access [85]. These works are promising, but further investigation remains before complete standardization.

A smaller and more difficult area for additional metrics to improve the initial scheduling process may be in characterizing the application to be run and matching its particular characteristics in use of graphics resources. There are several difficult aspects to this, including determining the requirement of the user for the application and analysing the nature of the graphics work with regard to the many different hardware structures there are (for example, in piping texture data versus geometry data around a network). Increasing diagnostic abilities in operating systems has recently been observed, which will make future analysis of applications and their typical utilization easier to achieve. A specific example is the DTrace component of Solaris 10 from Sun Microsystems. Although aimed at debugging problems and performance of any code, a side-effect of DTrace use is the ability to produce statistics that would, for example, be able to show how many *gl* calls had been made by an application during a particular duration. As these are the instructions that a Chromium session has to deal with, it would be a very relevant piece of data, particularly when correlated with other data on program execution.

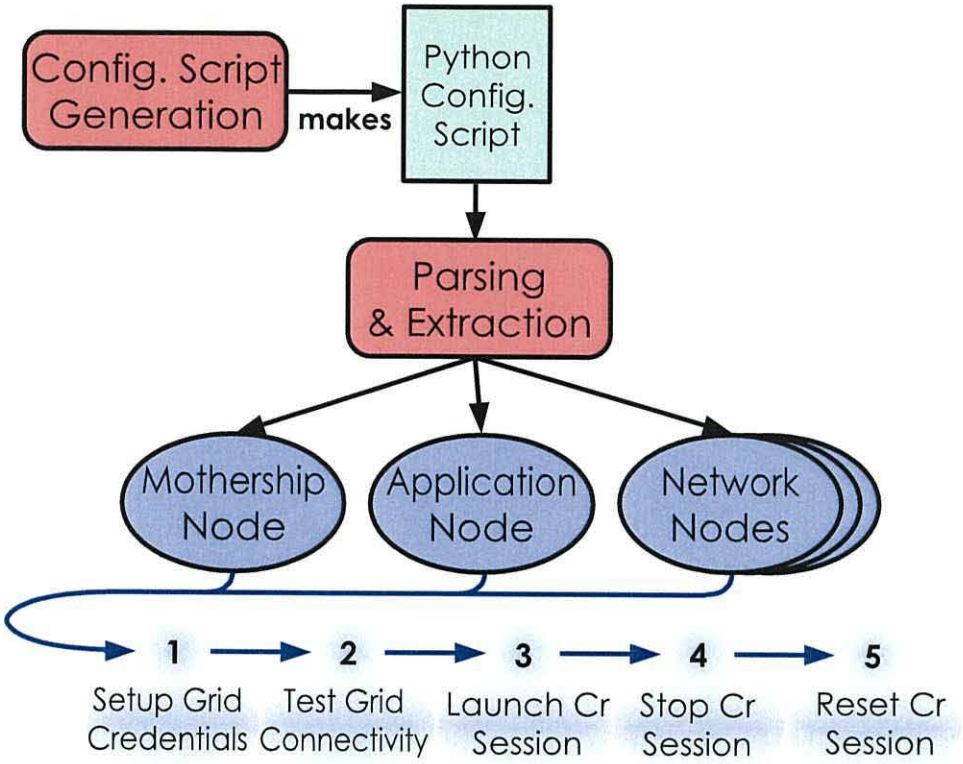


Figure 5.6: Components of jgViz runtime model

5.3 Conclusions

This chapter has detailed many of the defining and special features of the jgViz system along with the reasoning behind the design choices made. As detailed in figure 5.6, the runtime component of jgViz consists of a number of stages, some of which are susceptible to external influencing factors. Graphics resources present a subset of these regarding security of access to and performance in readback operations. These place restrictions on where and how such a tool as jgViz is useful. There are also performance restrictions implicit in the Grid protocols, as discussed regarding data streaming performance and the desirability of suitable protocols in order to ease security in multi-institutional Grids.

The pipeline of tasks involved in launching and stopping a graphics pipeline utilizes the GRAM, GridFTP and GASS protocols directly for job launching and data transfer. The particular characteristics of the two data transport protocols is seen to suit them better to either file transfer or low-intensity data streaming. We have found that the two monitoring schemes we studied have uses in different parts of the jgViz system. Due to the intensity of work involved, the round-robin node monitoring scheme is better for use at the scheduling

stage, but the concurrent scheme is necessary for effective monitoring of the nodes involved in a running pipeline. Regarding the monitoring stage, we base our current measurement scheme on the two metrics of network latency and node proportional load, and the comparison of short-term and long-term historical averages for this metrics. When the necessary deterioration in performance is seen, an event is triggered that stops the running pipeline and starts a new pipeline, along with limited preservation of state data.

Chapter 6

Experimentation

6.1 Introduction

Measurements taken for establishing whether jgViz meets the requirements originally expressed for it are an important part of understanding the nature of a distributed, Grid-based graphics pipeline. Additionally, jgViz's functionality in enhancing the distributed graphics pipeline experience can be studied to help determine areas that require further attention and where the focus for major future work should lie.

6.2 Experimental Setup

To exercise the variables that can apply to a jgViz distributed graphics pipeline's operation, we need a variable quantity of machines, a variety of network speeds and to test both types of distributed graphics pipeline. A temporary infrastructure was set up to build a suitable test environment. Figure 6.1 shows the machines used for this testing and referenced (by name) in this thesis alongside their locations in Bangor - this is the Bangor Grid.

Test Equipment

As shown in figure 6.1, the Bangor Grid consists of a number of different classes and types of hardware. The routers involved here (shown in blue), are of two types. The *sees.bangor.ac.uk* domain (Informatics) only has software routing (light blue) deployed on modest Sun Ultra 10s (ares, argo) that act as file servers and one Sun Blade 2000 (ajax)

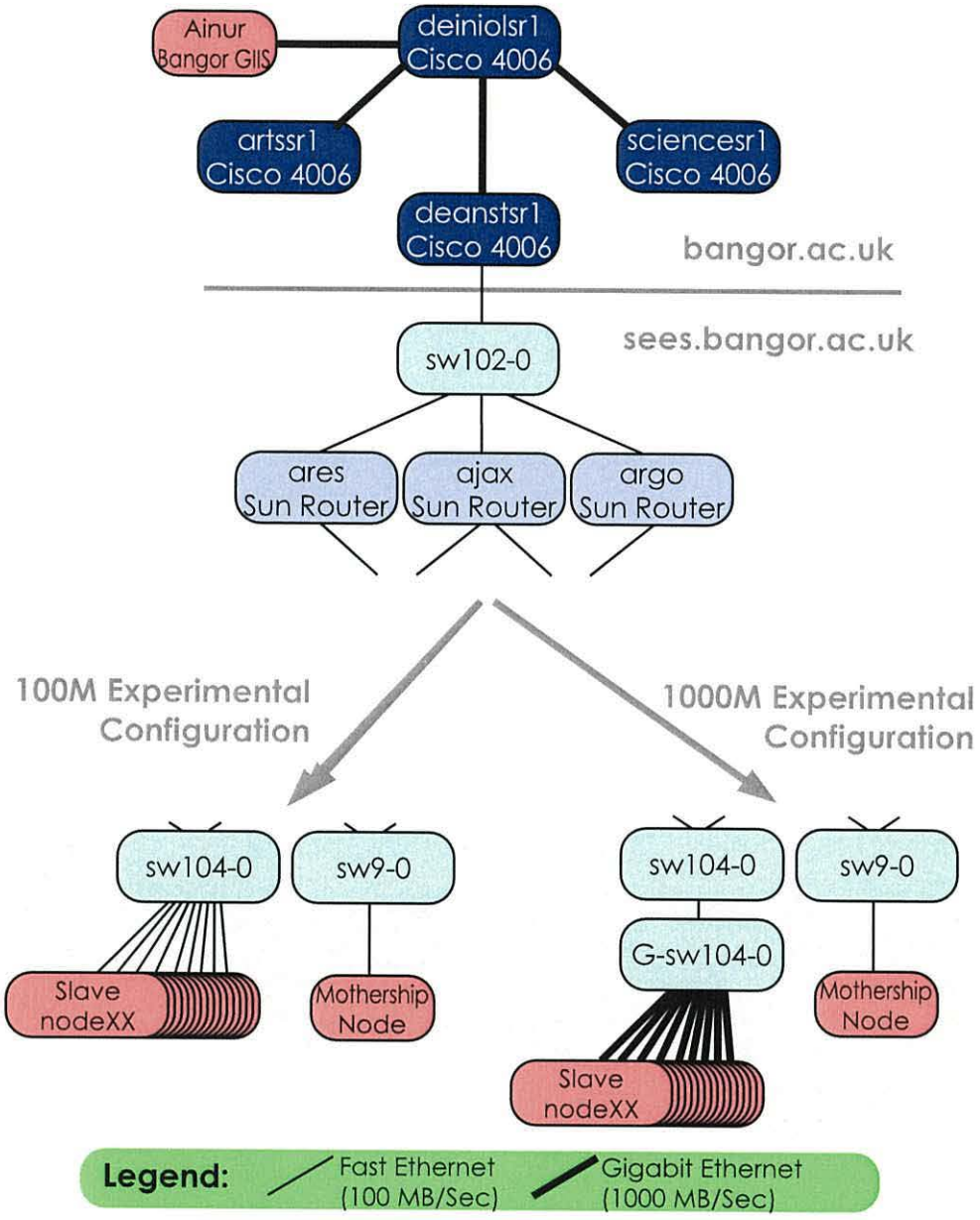


Figure 6.1: The Bangor Grid

that acts primarily as a software-installation server. It can be seen that *ajax* is a router for three subnets whereas *ares* and *argo* are for just two. The dark blue routers in *bangor.ac.uk* are Cisco Catalyst 4006 hardware switch-routers providing the university's Gigabit backbone and the connection between Informatics and *ainur*, the SGI Altix that acts as the Bangor GIIS.

The main experimental equipment consists of 18 PCs located in one of our laboratories, on subnet 147.143.104.0/24 - all of which are standard PCs with AMD Athlon-3200+ CPUs (actual clock 2.2GHz) and 1GB memory (see Figure 6.2). The rendering slaves are named *nodeXX* where XX is from 1 to 18. All but one of these machines have ATI 7000 series graphics cards (which do not currently support hardware rendering under Linux) and the remaining one (to be used as the compositor in readback) has an nVidia GeForce 3, which does support hardware rendering, thus somewhat lightening the load of intensive graphics operations. The experimental machines all run Fedora Core Linux 4 and the X.org 6.8.3 X server. The *mothership node* is a P3-1GHz, 256MB RAM PC running Red Hat Linux 9 and XFree86 4.3.0. *ainur* is an Altix 3000 with 12 Itanium2 CPUs at 1.3GHZ and 12GB RAM running SGI ProPack Linux 3 (Red Hat Enterprise derived). All machines involved in the Bangor Grid use the Globus Toolkit 2.43 and run GRIS servers advertising their abilities to the GIIS. Chromium 1.8 was utilised on all visualization machines. nVidia's Linux driver version 1.0.7664 (supporting OpenGL 1.5.3) was used where appropriate and Tungsten Graphics' MesaGL 6.2.1 DRI Radeon Driver (supporting OpenGL 1.2) used on the other machines.

Our experimental setup connects the slave test machines at one of two speeds - Fast Ethernet 100MB/Sec using an in-room 3Com 4300 switch (*sw104-0*), or Gigabit Ethernet 1000MB/Sec using a Cisco Catalyst 4006 for switching (*G-sw104-0*). This Gigabit switching is actually carried out by the *deanstsr1* Catalyst 4006 with appropriate ports on it made into a separate logical partition through Virtual Local Area Networks (VLANs). The amount of traffic otherwise handled by *deanstsr1* is so low that we consider there to be no possible side-effects of this dual role for the one physical box - 6 months of SNMP monitoring failed to produce a utilization figure (in CPU or switching bandwidth) above 4%. The major traffic occurs solely within the confines of subnet 147.143.104.0/24, so does not additionally suffer from the performance of the links in the Informatics network and backbone (147.143.102.0/24). The gigabit switching is carried out over 50 meter long cables from the experimental machines, but all connections were tested complete and showed matched, consistent performance and no errors. A brief discussion of whether long cables makes a difference is included in Appendix A.

Figure 6.2: Experimental Setup showing Rendering Nodes (see also page 172)

Test Application

Experimental performance is always going to be affected by the application chosen to demonstrate that performance. We are restricted by the experimental hardware available, but our absolute need is to provide a representative and real-world study. In particular, our main experimental apparatus does not support hardware rendering of OpenGL data and this will restrict the performance obtained. We also have a requirement to establish good baseline performance so that the implications of changes to variables can be well observed. As a result, we have chosen a test application that involves relatively little texture data and places variable geometry demands on the graphics pipeline. The application is *Roller Coaster 2000* by *Plusplus* [86], which is a C/OpenGL roller coaster ride animation in which the roller coaster tracks are drawn using different levels-of-detail as per their distance from the 'camera'. The roller coaster tracks are mathematically defined by control points and tangents, and so are programmable and repeatable. The speed variation (within the simulation) and variable rate of graphics commands will produce an ideal situation for monitoring the pipeline performance in both situations and for observing effects such as stutter. The jgViz system should be able to support any OpenGL 1.5 compliant application due to Chromium's API compatibility. Community experience shows, however, that some applications do things in unexpected ways and require some 'tweaking' in order to make them work.

Test Conditions

All experimental results are taken in a fully-deployed jgViz system. All resource discovery is performed using the Grid Index Service, all job launching is carried out through the Globus Resource Allocation Manager, all file transfer with the Grid FTP service, all graphics work is handled through a standard Chromium installation and all user interaction is with the jgViz client. Measurements are taken multiple times in two separate sittings. Variation in results obtained, such as frame rate, is alleviated by the measurements being taken at the same set stages of the looping Roller Coaster demonstration run, with the default 'rc2k.trk' roller coaster track, and then averaged.

6.3 Experimental Procedure

There are many measurements that can be taken when involving multiple machines in a distributed manner. There has been work in measuring the performance of Chromium style

graphics pipelines, including in [16]. However, testing the jgViz system itself is more difficult due to the Grid middle-layer architecture and the very fact that part of jgViz is aimed at alleviating some of the possible performance problems in such an environment. So, we focus on evaluating the additions to the process of using distributed graphics pipelines provided by jgViz. Effectively this is the scalability of the jgViz setup system - specifically, the parts that initiate graphics pipelines and the parts that then take responsibility for keeping them healthy. In establishing this, there are two principal metrics. Firstly, the *monitoring cycle time*, which is the time to complete a single monitoring run of all pipeline nodes, given by instrumenting the monitoring thread (see Appendix B for a discussion of the GUI) to record the time when it begins to update its load and ping metrics for each node, and then to record and calculate the time duration taken after a single update of each node. The *monitoring cycle time* reflects on the 'behind-the-scenes' quality of whether jgViz is able to assess the pipeline nodes frequently enough, and thus trigger an event quickly when the pipeline needs attention. Secondly, the *reschedule time*, which is the time duration between an event triggering to show a pipeline as no longer operating optimally, and it being stopped, rescheduled (including updating node metrics) and restarted on a new set of machines. This value measures the practicality of the jgViz monitoring/rescheduling system. Judging whether the stage in between these two metrics - the decision of whether a pipeline is 'good enough' - is adequate in itself is not straight forward. This will be tackled as a process of optimizing the monitoring parameters jgViz makes available, although fine-tuning may still be needed for a particular, different, network infrastructure. Additionally, in assessing the broad-picture performance, the *frame-rate* is relevant. At a more technical level, CPU, memory and network statistics will be taken for each of the types of nodes involved in a pipeline to allow us to analyse where the limitations lie. The four types of nodes are:

The client node - that on which the jgViz client runs.

The application node - that on which the application is hosted by the Chromium application faker.

Rendering slave nodes - those that take in GL commands, perform them (i.e. render them) and then either display the results locally (tiled pipelines) or read back the results and send it over the network to a compositor node (readback pipelines).

The compositor node - the node that takes in multiple rendering streams, reassembles them and displays them locally (readback pipeline only).

The low level metrics are as follows:

CPU User Time (%) - percentage of CPU time spent on user calls.

CPU System Time (%) - percentage of CPU time spent on system calls.

CPU Wait IO Time (%) - percentage of CPU time spent waiting on Input/Output sub-systems.

CPU Idle Time (%) - percentage of CPU time spent idling.

Memory Used (bytes) - amount of memory used over and above base Operating System level.

Network RX (bytes/second) - bytes received per second via the Ethernet network.

Network TX (bytes/second) - bytes transmitted per second via the Ethernet network.

6.4 Base Results

6.4.1 Monitoring Cycle Time

As with the previous comparison of round-robin and concurrent monitoring strategies (see chapter 4), we have instrumented the jgViz client's pipeline runtime code to output a number of statistics, including the monitoring cycle time. It is particularly important that this metric scales well, due to the fact that the entire rescheduling process's efficiency depends on it.

The results for tiled and readback pipeline configurations (shown in figures 6.3 and 6.4) are, understandably, fairly similar. In both cases, we see that Gigabit Ethernet scales better as the node count increases. The higher low-end result is due to the fact that a readback pipeline implicitly has one more node to be monitored (the compositor node) and that this node would typically be busier than any of the slave network nodes. It is also noteworthy that the Fast Ethernet results scale very-close to linear, suggesting that a network overhead does not impinge at the low-end in the same way that it seems to with Gigabit Ethernet, where the scaling is a definite shallowing curve. Broadly, these results are independent of the network speed due to the low volume of data being handled. It is more likely that contention for the network due to coincidences in timing (i.e. two monitoring requests

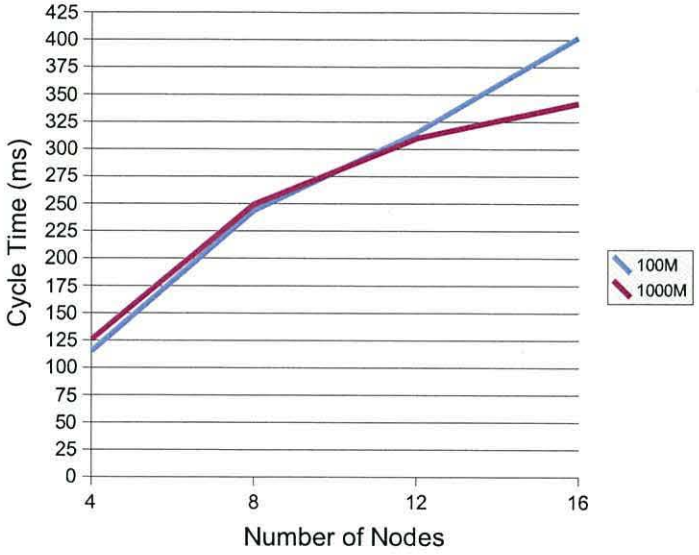


Figure 6.3: jgViz Monitoring Cycle Time for Tiled Pipeline over Fast and Gigabit Ethernet Networks

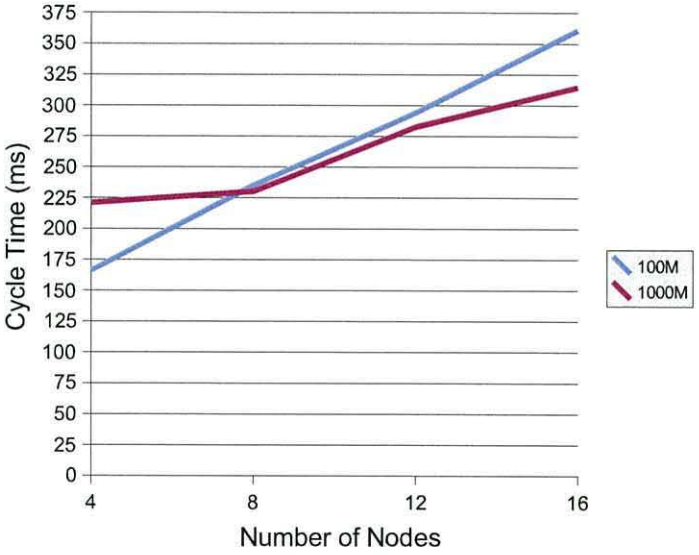


Figure 6.4: jgViz Monitoring Cycle Time for Readback Pipeline over Fast and Gigabit Ethernet Networks

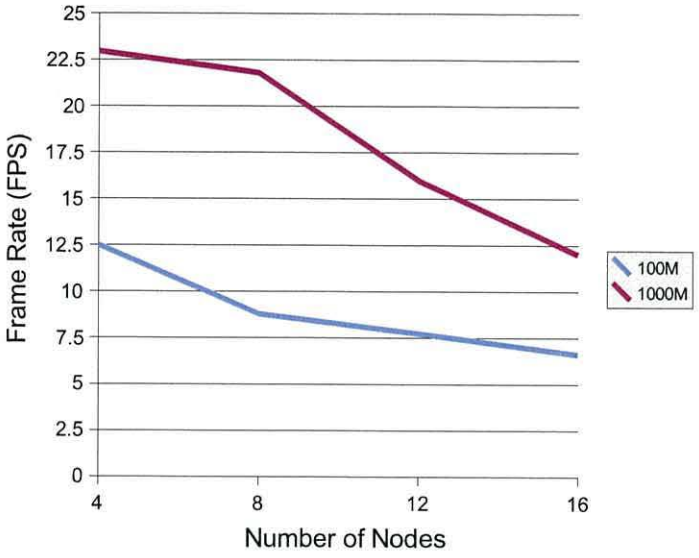


Figure 6.5: jgViz Frame Rate on Tiled Pipelines over Fast and Gigabit Ethernet Networks

being sent for different machines simultaneously) would be a problem. This may explain why the Gigabit result scales better to 16 nodes as it takes less time to send the small amount of data in the request (compared to Fast Ethernet), and so is less likely to coincide with another request being sent.

6.4.2 Frame Rate

As previously mentioned, the frame rate is certainly relevant to the operation of a visualization and so deserving of attention here. The specific details of taking such a technology and putting into a distributed computing environment means that we will see less performance and more variation than in a more tightly-knit setup.

Providing a consistent and fast enough frame rate is important for interactive visualization. This is always going to be difficult to achieve within a distributed computing environment.

Frame Rate is particularly sensitive to change in the two major variables of pipeline type and network speed. Of interesting contrast shown in figures 6.5 and 6.6, however, is the differing trends seen. Tackling tiled pipelines first, both Fast and Gigabit Ethernet networks provide reasonably consistent drop-offs in performance as the number of nodes increases. The shallower drop-off in Fast Ethernet performance indicates that the network is becoming a limiting factor at the application node (where the scene is split and sent to

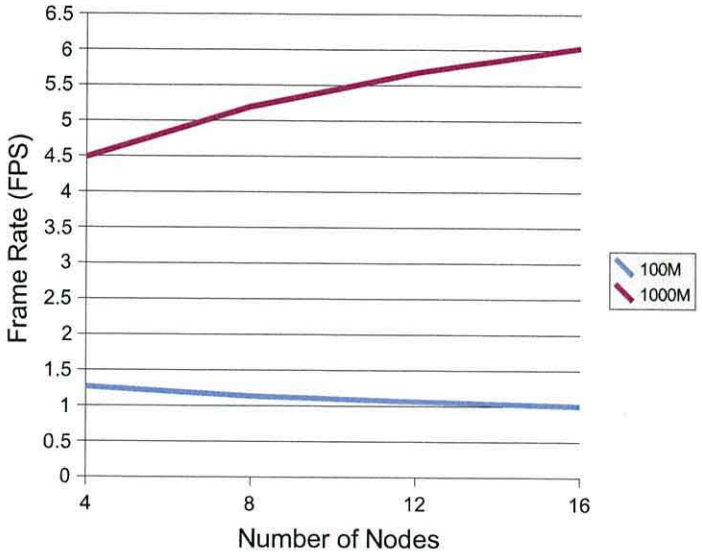


Figure 6.6: jgViz Frame Rate on Readback Pipelines over Fast and Gigabit Ethernet Networks

the render slaves) at a fairly low number of nodes (i.e. the 100Mb/Sec available bandwidth becomes saturated, even though the machine itself can provide far more data). Gigabit Ethernet results show more severe drop-off although whether this is due to network or node limitation is unclear. The difference between Fast and Gigabit performance is also observed, and Gigabit certainly gives acceptable performance. Readback results are interesting. Here, the difference between the two network technologies is more pronounced. Fast Ethernet performance starts poor and gets worse. Gigabit performance is far from good, but actually improves as more nodes become available. This is indicative of network limitations, but in a more distributed manner than in the tiled pipeline case. In considering results for any rated speed of network, we need to remember two things - that the actual performance is likely to be different, and that performance over a duration of a second is not defined well enough for a visualization running at a rate in excess of several frames per second. However, the performance of Fast Ethernet readback being so close to 1 frame per second is indicative of a clear network limitation. Gigabit performance shows slight improvement with increasing number of nodes, indicating that the lowering amount of work being done by each render node reduces the overhead of data that has to be transferred between any two hosts (render slave and compositor). As an overall view, it seems fair to conclude that readback performance would benefit greatly from improved network performance, both in latency (where each frame component is subject to the network latency twice - once from application node to render slave and then from render slave to compositor) and bandwidth

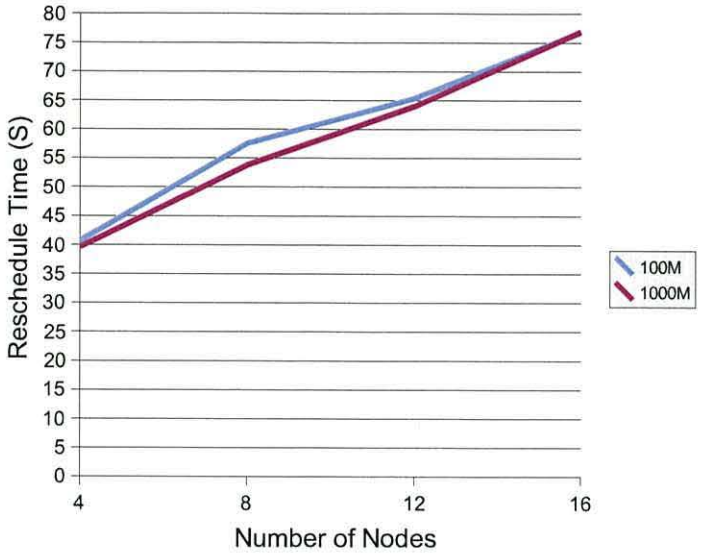


Figure 6.7: Pipeline Reschedule Time of Tiled Pipelines over Fast and Gigabit Ethernet Networks

(getting as many frame components from a number of render nodes to a compositor node all as simultaneously and instantaneously as possible).

6.4.3 Pipeline Reschedule Time

One of the fundamental jgViz functions is to provide reliability and consistency in the pipeline through monitoring and rescheduling of it. Rescheduling presents its own problems, particularly in a distributed Grid environment trying to provide interactive, real-time graphics. An overhead of all the security components in the Grid protocols is that job launching takes a certain amount of time. There is little that can be done to reduce this without compromising the protocols themselves. Parallelism in job launching may help, but the balance of processing and communication required by the security components of job launching limit the use of this approach without increasing the jgViz client machine capacity. So, this is an important metric of jgViz's performance as it references directly how jgViz itself works.

Reschedule time scales linearly as we would expect. In both tiled and readback setups, there is approximately 40 seconds of overhead, as shown in Figures 6.7 and 6.8. This is due to the stopping of the old pipeline that is running (involving stopping the mothership and killing the application via Grid jobs), the time spent refreshing metrics data on available nodes,

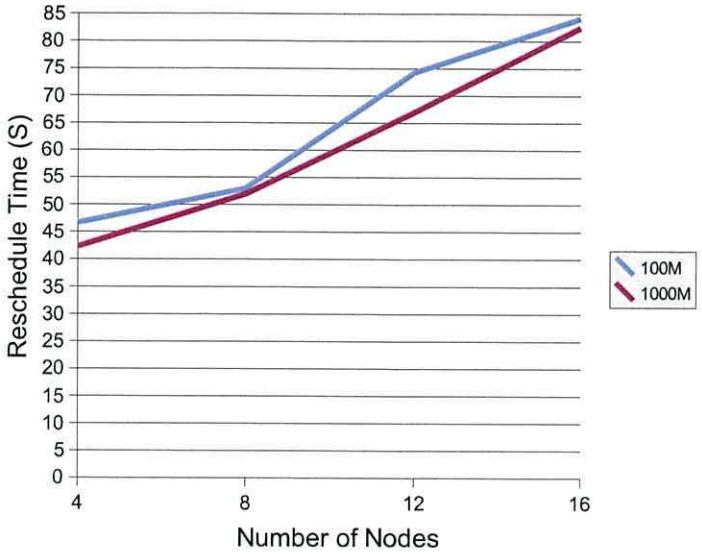


Figure 6.8: Pipeline Reschedule Time of Readback Pipelines over Fast and Gigabit Ethernet Networks

transfer of the new mothership configuration to the new mothership and the launch of both the new mothership and application nodes. The time taken for these tasks to be carried out remains broadly constant as this 40 second period. The additional few seconds on this initial overhead in a readback pipeline is due to the additional compositor node being launched at each scale, no matter how many network nodes are involved. The differing results then seen on top of this are for different numbers of network nodes to be launched. The slight disturbance seen in the Fast Ethernet results relative to the very-consistent Gigabit results is due to the fact that the nodes involved in the pipeline are more likely to be network limited with Fast Ethernet.

6.4.4 Low-Level Statistics

Having discussed some of the higher-level performance results and speculated over possible causes thereof, it is appropriate to investigate some of the low-level causes of this behaviour and, thus, better characterize the issues faced in distributed graphics pipelines. We begin by observing the CPU utilization statistics for the various node types with the two network speeds as part of the tiled pipeline type.

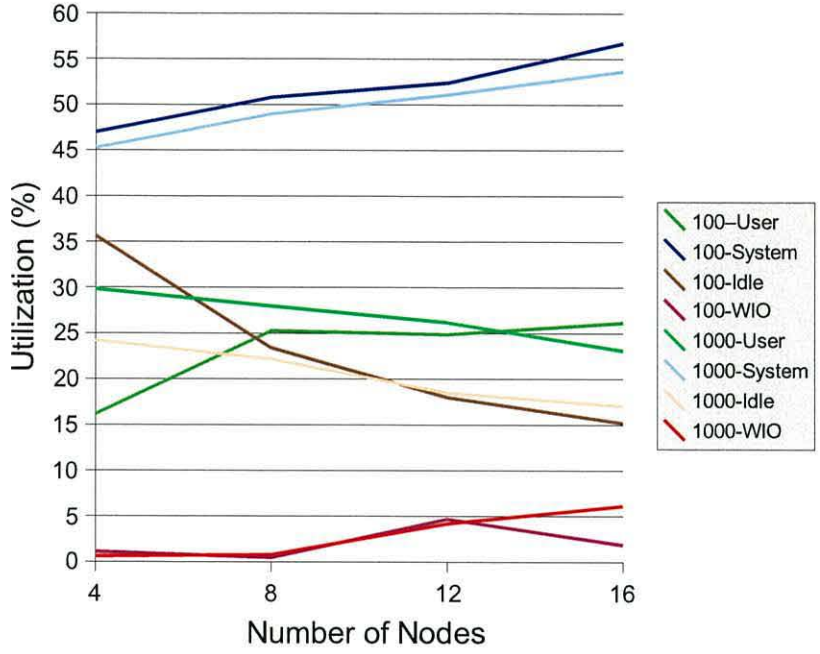


Figure 6.9: CPU Statistics for Client Node Monitoring Tiled Pipeline over Fast and Gigabit Ethernet Networks

CPU vs Network Speed

The client node gives reasonably predictable results, as shown in Figure 6.9. As the number of nodes being monitored increases, the idle time reduces and the wait IO time increases, reflecting the increased network traffic being seen. Likewise, the system time increases and, at its much more significant 45% levels, reflects just how tied to network communication the jgViz client's monitoring is. That the user time remains reasonably consistent implies that the jgViz code itself scales reasonably well, but that the communication overhead restricts the test environment to monitoring some 35-45 nodes.

The application node is responsible for splitting up the graphics scene and sending it out to the render slaves. Figure 6.10 shows the differing user time behaviour in Fast and Gigabit networks. The limitations of the 100MB/sec network for more than four nodes actually reduce the user load on the application node as it is, in effect, required to generate less frames (and thus do less work) in order to keep the later-stage 'pinch-point' fully occupied. In contrast, we see that the 1000MB/sec network provides increased capacity for the pipeline beyond four nodes, with reducing idle time and increasing wait IO and system time. This would lead to a limitation of up to 30 nodes or so in this network environment.

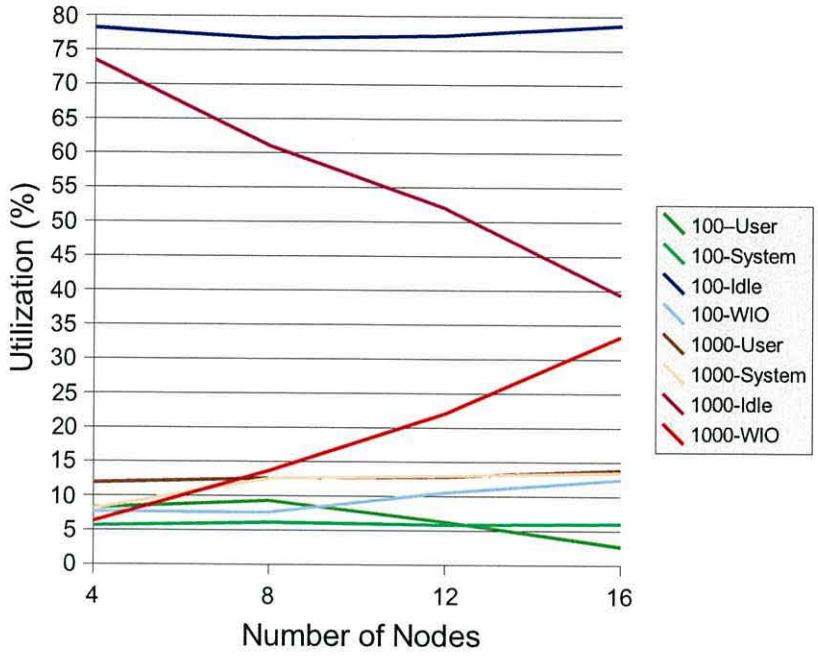


Figure 6.10: CPU Statistics for Application Node as part of Tiled Pipeline over Fast and Gigabit Ethernet Networks

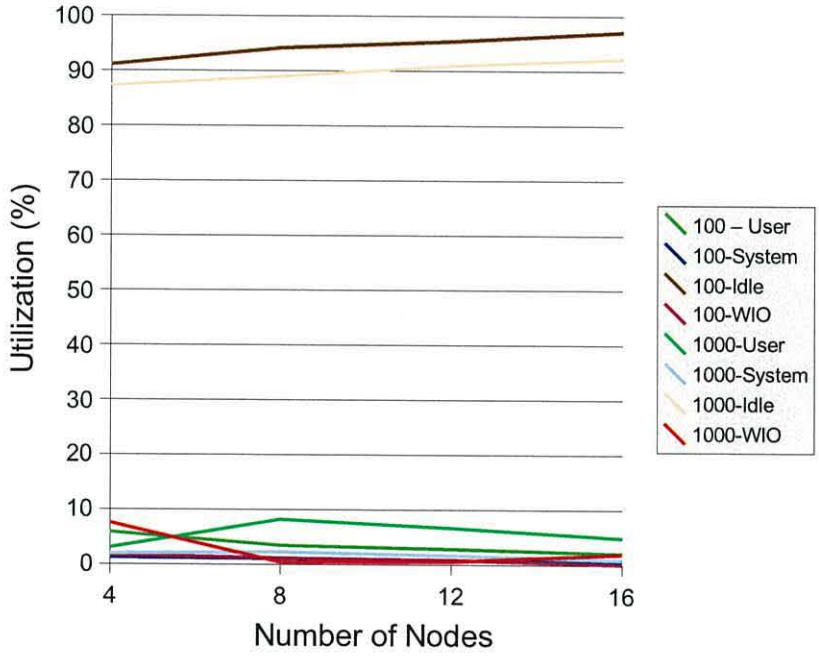


Figure 6.11: CPU Statistics for Slave Rendering Node as part of Tiled Pipeline over Fast and Gigabit Ethernet Networks

The statistics for the individual slave rendering nodes, seen in Figure 6.11, show the idle time for each node increasing as more nodes are made part of the pipeline. This is due to the fact that each node has less overall work to do as it receives a smaller chunk of the scene to render and the frame rate lowers. The relative difference between the idle time and the other three CPU metrics implies that a more complicated visualization could easily be used. However, our test equipment would not suit this well due to the software GL layer used on the slaves through MesaGL, and the fact that we may then restrict our ability to observe effects on the application and compositor nodes.

CPU vs Pipeline Type

To observe the effect that the pipeline type has on CPU utilization, we study the relevant nodes (with the extra node of the compositor in a readback pipeline) in a Gigabit network only. Previous observations make it clear that doing so in a Fast Ethernet network would not produce truly realistic figures for the CPU due to the network bottle-neck. Whilst the network may still be the overall limiting factor with use of Gigabit Ethernet (due to it as a technology rather than saturation in a certain part), the scalability can be shown alongside actual performance data.

The effect of monitoring a readback pipeline as opposed to just a tiled pipeline is largely that of simply monitoring an additional node (see Figure 6.12). System time increases with more nodes and idle time reduces whilst user time remains very similar, reflecting increased network utilization. Wait IO time is insignificant in both pipelines and, indeed, it should not vary for the client node.

The application node (see Figure 6.13) gives very different scalability characteristics. In terms of idle time, it can be seen that a tiled configuration consistently decreases whereas a readback configuration decreases and then increases again as the number of nodes involved rises. This is due to a bottle-neck further down the pipeline that only becomes apparent with the extra requirements of a readback pipeline over a tiled instance. It can also be seen that the idle and wait IO times are closely linked for the tiled and readback pipelines. System and user time in both pipelines increase linearly and without note, as we would expect as the amount of data the application node is generating, splitting and sending does not radically change in size, only in the way it is partitioned.

The slave render nodes (Figure 6.14) show increasing idle time in those pipelines with more nodes, causing a drop in frame rate. That readback really suffers with the increasing node-

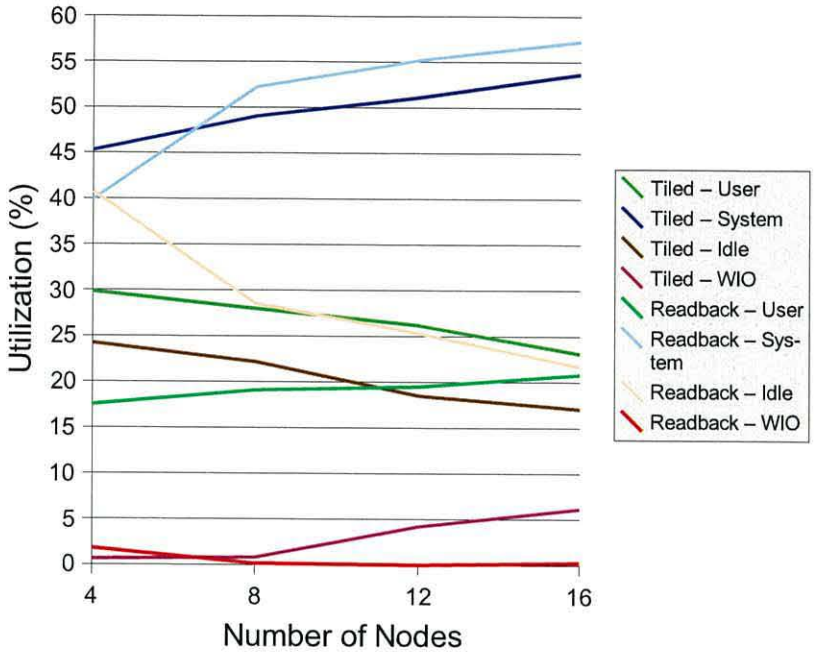


Figure 6.12: CPU Statistics for jgViz Client Node monitoring a Gigabit Ethernet Network providing both Tiled and Readback Pipeline Types

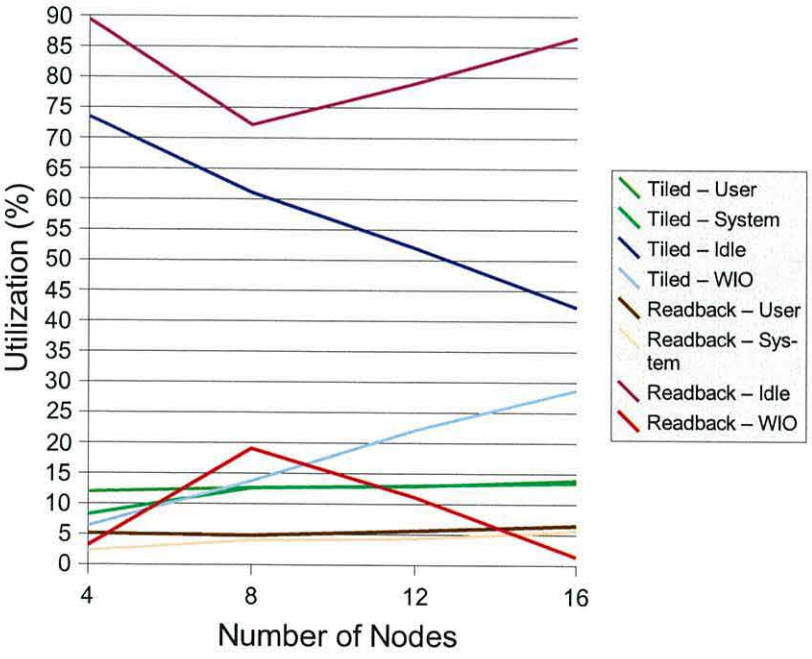


Figure 6.13: CPU Statistics for an Application Node as part of a Gigabit Ethernet Network providing both Tiled and Readback Pipeline Types

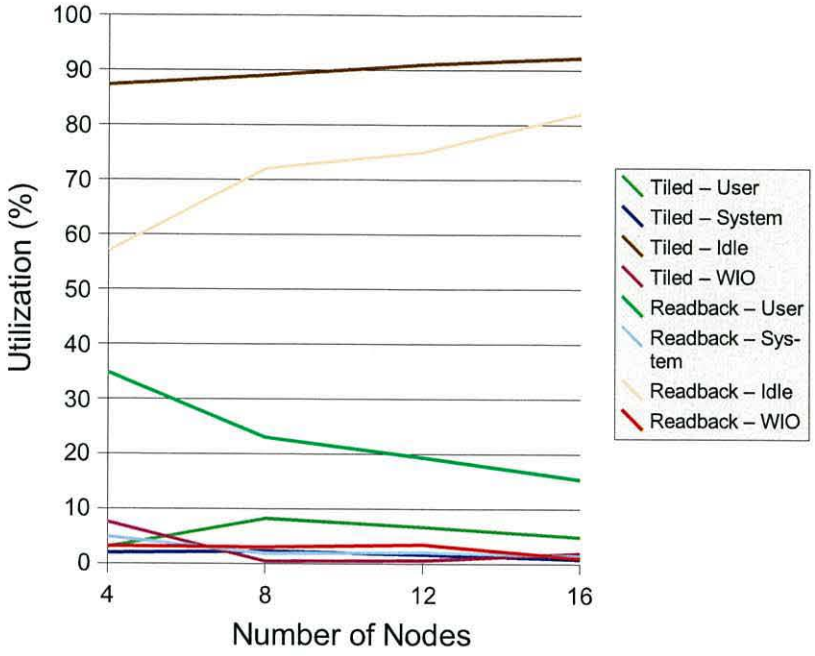


Figure 6.14: CPU Statistics for a Slave Node rendering as part of a Gigabit Ethernet Network providing both Tiled and Readback Pipeline Types

count is clearly shown by the peak in user time being at the lowest number of nodes. This sharp fall is in contrast to the tiled setup in which user time peaks at 8-nodes and only suffers a steady drop-off after that.

Compositor performance (Figure 6.15) shows peak-efficiency at the 8-node count. It is also clear from the time spent in wait IO relative to the time spent in user and system processing, that the network latency is a problem here. It seems reasonable to suggest that the hardware graphics card present in our compositor node is responsible for part of the lowness of the load here and, as we have seen that the rendering slaves are not bottle-necks above, and that the network is the cause of the slow down in the pipe in getting the rendered tiles back to the compositor quickly enough. This would concur with the high wait IO time observed, which is caused by transmission of data.

Network Traffic

As previously observed, there are two types of limitations with the network. Firstly, there is the capability of the network technology standard available. For example, there is an

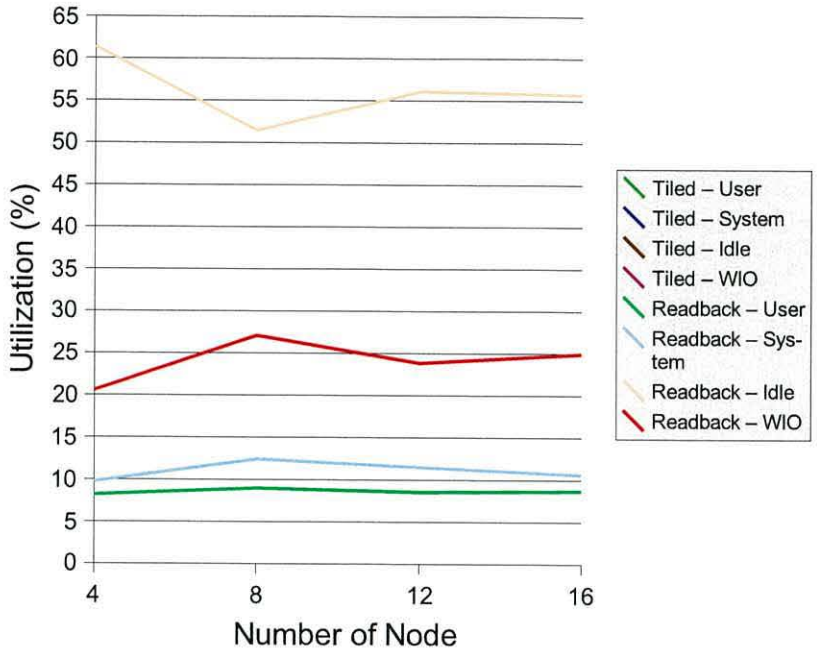


Figure 6.15: CPU Statistics for the Compositor Node in a Gigabit Ethernet Network providing both Tiled and Readback Pipeline Types

implicit latency overhead associated with any network technology. Secondly, there are the limitations of any network environment, including cabling and switching. Having a Gigabit Ethernet network does not guarantee that we will be able to transfer 1 Gigabit of data per second from or to a machine. In addition, it is worth noting that a graphics pipeline is only as strong as its weakest part and that, with a distributed setup, this places a great requirement on the networking involved in connecting the pipeline components.

We have performed some initial testing with *netperf* [1] that show the data transfer levels we have been able to get, as shown in table 6.1. The bandwidth tests represent sustained transfer between three different sets of machines in the slave test group for durations of one hour. We have also used ICMP Echo packets via *ping* to establish network latency over an hour-long duration.

[1] `http://www.netperf.org/netperf/NetperfPage.html`

Network Technology (Mbits/Sec)	Real-World Speed (Mbits/Sec)	Latency (us)
Fast 100	93.93	124
Gigabit 1000	269.00	93

Table 6.1: *netperf* and *ping* measured real-world network data transfer rate

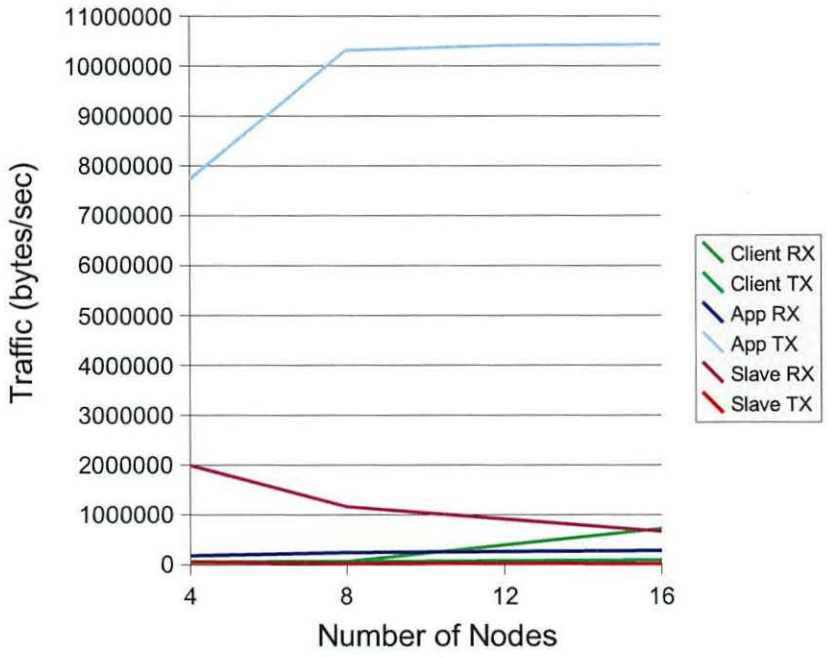


Figure 6.16: Network Traffic seen by all node types in a Fast Ethernet Network providing a Tiled Pipeline

These results tie in with what we would expect given the differences between the two Ethernet networking technologies. The bandwidth achieved by the Gigabit link is somewhat less than we would expect, but is still large enough to not be a problem with the number of nodes we have available and is sensible. Going above 16 nodes produces a situation where other limitations regarding processing the graphics stream come to the fore - such as performing the computation involved in partitioning a scene into multiple tiles. At such node levels, limited bandwidth is not the problem we will encounter. In attempting to understand the values regarding latency, we have performed a short study of the Fast and Gigabit Ethernet standards in Appendix A.

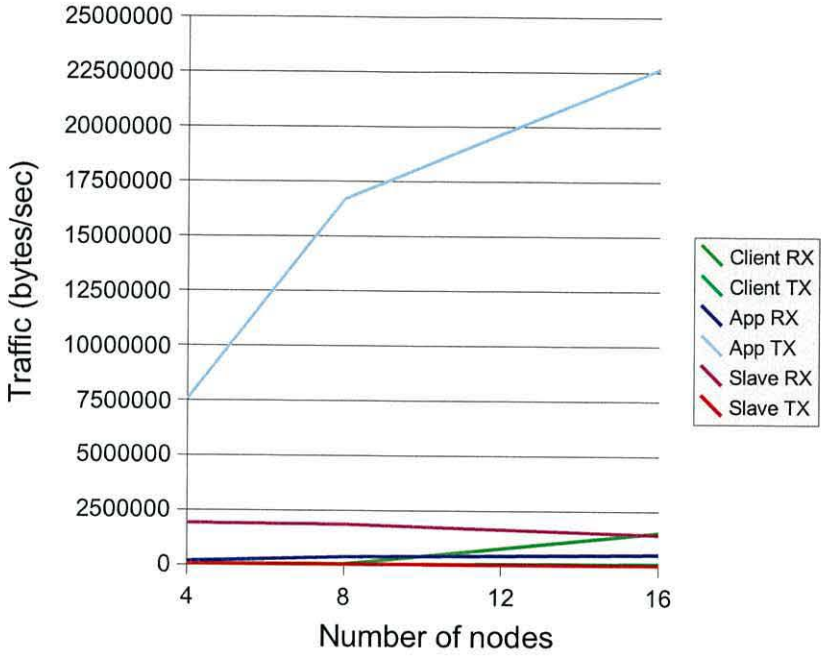


Figure 6.17: Network Traffic seen by all node types in a Gigabit Ethernet Network providing a Tiled Pipeline

Figures 6.16 and 6.17 present the useful contrast of the two network technologies we have. We note that the Fast Ethernet results indicate that the network at the application node becomes saturated with a fairly low number of nodes being sent to by the application node. The Gigabit results allow us to see that this occurs at approximately 5 nodes and that the increase in the applications node's transmitted data begins to slow down at 8 nodes. At the same time, we see that traffic being received by the render slaves is reducing. This is particularly apparent with Fast Ethernet as the same amount of bandwidth is split between increasing numbers of nodes, but is also apparent with Gigabit. These statistics broadly agree with the results seen for CPU statistics (figures 6.12, 6.13 and 6.14) with the application node idle time reducing and then beginning to ever-so-slightly level off. We can easily determine the area of peak data transfer in this pipeline as being the transmission from the application node to the render slaves.

With this knowledge in hand, it seems reasonable to hypothesise two things: firstly, that the Gigabit tiled pipeline's application node was beginning to encounter delays from the network as its bandwidth utilization approaches the real-world limits - 22.5 MBytes/Sec = 180MBits/Sec - within the above mentioned idea that a second is a long time in a multi-frames-per-second visualization; and secondly, that the fact there is still some room

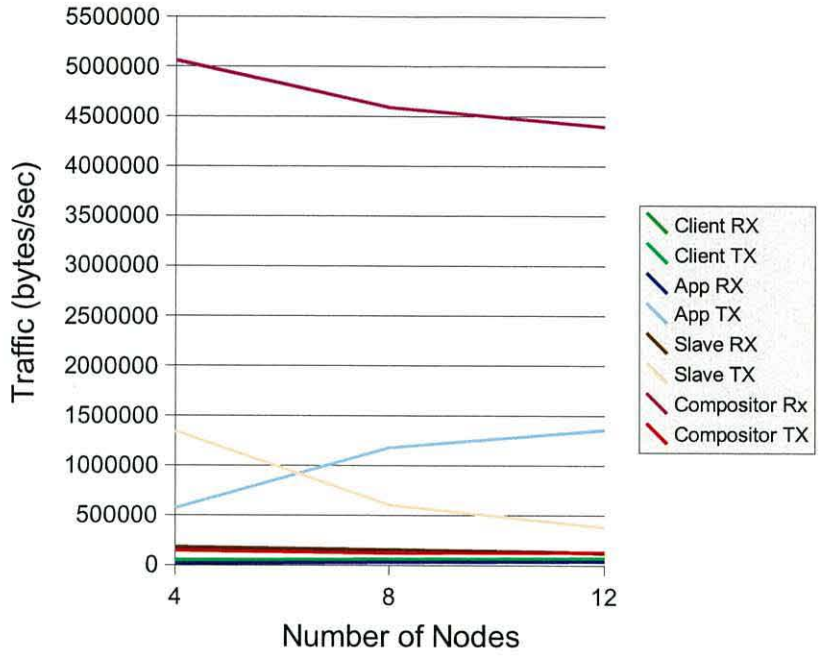


Figure 6.18: Network Traffic seen by all node types in a Fast Ethernet Network providing a Readback Pipeline

for increase implies the physical limitations of the system in non-network-bandwidth terms have been reached, as discussed above.

The readback pipeline results shown in figures 6.18 and 6.19 give very similar shaped results for both network speeds. The fact that the Fast Ethernet network is not nominally saturated (at less than 5.5MBytes/Sec) does not consider the dual transit over the network now being taken for each frame to be displayed on the compositor node. With the latency for each frame being incurred twice, the amount of data involved in transporting rendered pixels back to the compositor is a clear limitation in the very bursty data transfers implicit in a frame-by-frame visualization. A different application would give a different relationship between the application node's transmitted data rate and a render node's transmitted rate (i.e. geometry data vs pixel data) - in particular, the heavy use of texture data would have a large impact on its initial distribution from the application node. The higher bandwidth of the Gigabit network improves the peak performance to the 8-node marker from the much lower value in the Fast Ethernet network, but fundamentally lacks the performance to make readback truly interactive.

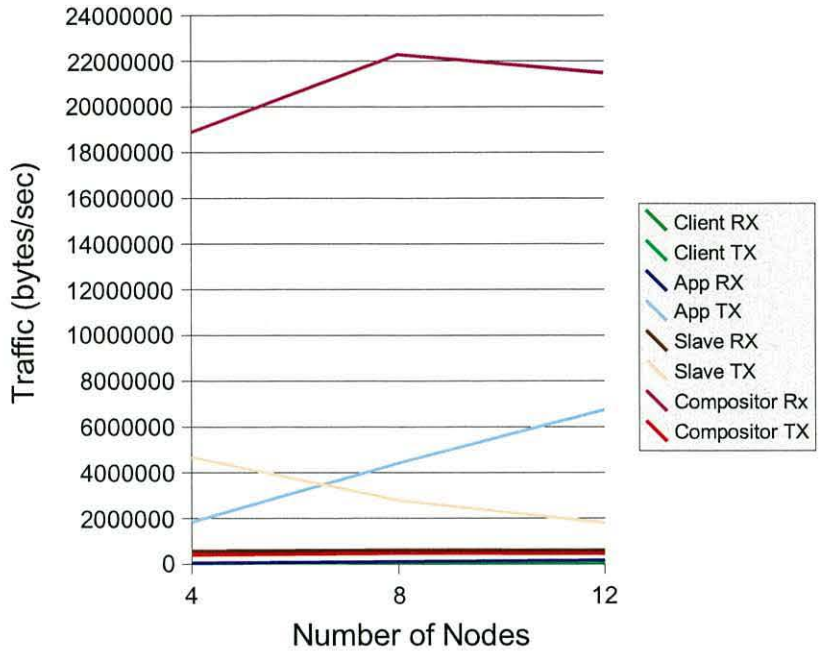


Figure 6.19: Network Traffic seen by all node types in a Gigabit Ethernet Network providing a Readback Pipeline

6.4.5 Network Congestion Effects

It is clear that jgViz distributed graphics pipelines are most sensitive to network issues. It is, therefore, prudent to look into the effects of congestion in different parts of the network. To establish an understanding of this, we have repeated the measurements from the previous section with network congestion in place. In order to establish as wide a view as possible, we have only carried this out for tiled pipelines, since that is where performance is not so lacking already that seeing the impact of the network congestion would be difficult. We use Gigabit networks and not Fast Ethernet, as the 100MB/Sec also does not provide enough performance. In order to congest the network, we simply use *netperf* on an additional (non-pipeline) machine and bombard the point of network congestion with a bandwidth test. We are not interested in the results of the test as we only require it to congest the network for the period of time when the application is running. We therefore run netperf with an arbitrarily large duration and quit it when we no longer need it. The network congestion simply takes the form of a high-traffic but low-overhead application. TCP was used rather than UDP, as it is imagined that most future applications will use TCP in a Grid as they are unlikely to be failure-tolerant. This produces some load on the processor of the machine in handling the sheer number of interrupts that will have to be serviced. However, having

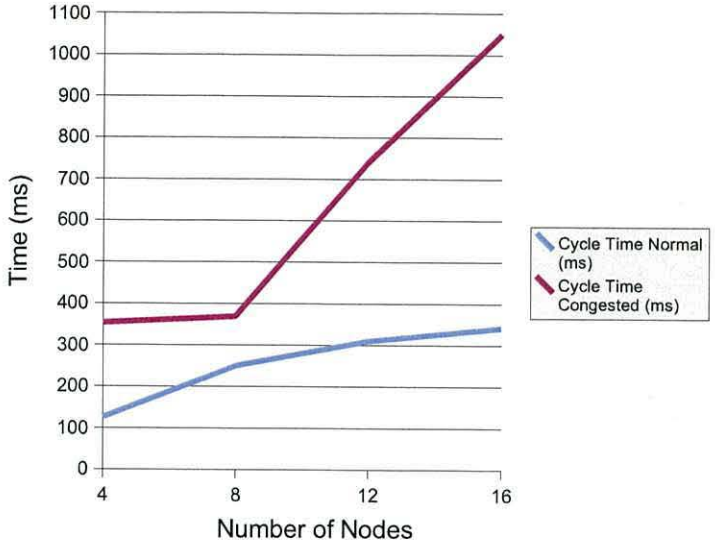


Figure 6.20: Client Monitoring Cycle Time in Gigabit Tiled Pipeline with Congested Client Node

processors as fast as the 3.2GHz-rated Athlons limit these problems, even if the front-side bus speed is so much smaller and no matter how the network software driver is setup to request interrupt service from the CPU.

Across all this new experimentation, it was observed that no noticeable memory differences occurred from the standard results above and that the statistics for nodes not being directly congested only varied in the downward direction. We therefore do not revisit those results already discussed as part of the non-congested experimentation.

Congesting the Client

The jgViz client machine in a pipeline does not usually receive a substantial amount of data. The level of utilization associated with monitoring the machines involved in the pipeline is relatively small.

Figures 6.20 and 6.22 show clearly how detrimental the network congestion can be to the jgViz client's main functions. The fact that the monitoring cycle time increases so dramatically represents the fact that it is a lighter load than the actual rescheduling process, which suffers significantly but consistently. Figure 6.23 shows us that the reason for this as being vastly increased system and wait IO CPU time proportions over the non-congested

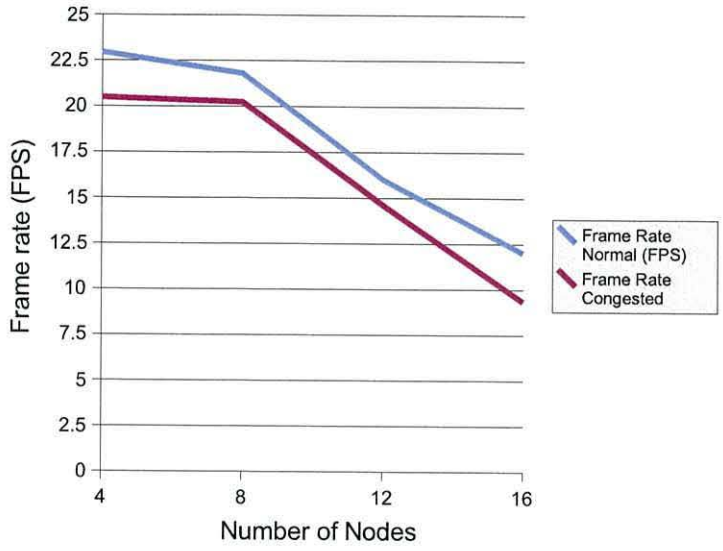


Figure 6.21: Visualization Frame Rate in Gigabit Tiled Pipeline with Congested Client Node

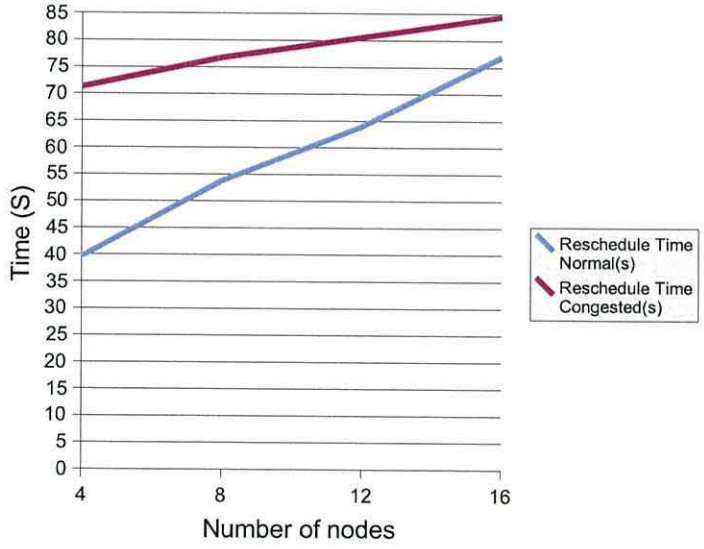


Figure 6.22: Reschedule Time for a Gigabit Tiled Pipeline with Congested Client Node

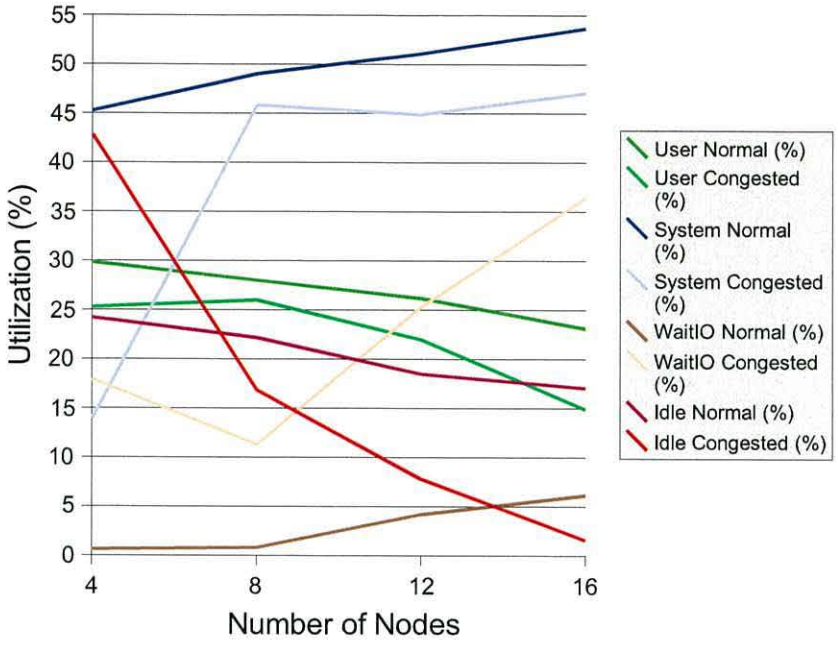


Figure 6.23: Client CPU Statistics in Gigabit Tiled Pipeline with Congested Client Node

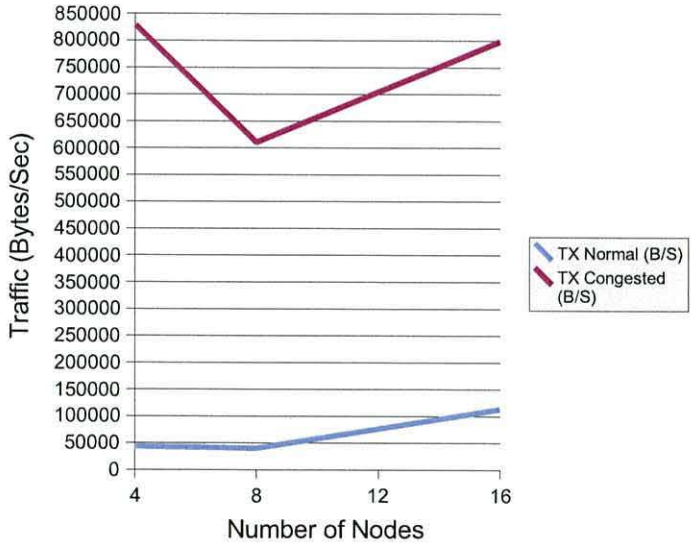


Figure 6.24: Client Transmitted Network Traffic in Gigabit Tiled Pipeline with Congested Client Node

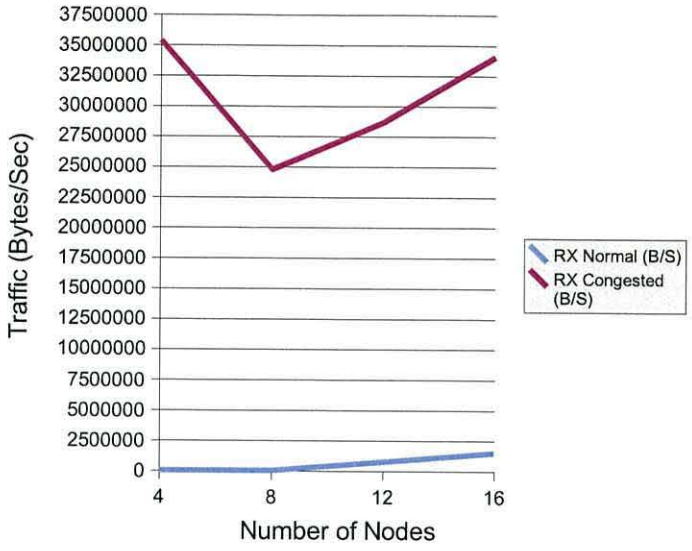


Figure 6.25: Client Received Network Traffic in Gigabit Tiled Pipeline with Congested Client Node

(normal) setup. The system time increase represents servicing of interrupts associated with the bombardment of incoming traffic whereas the wait IO time increase is associated with the delays now occurring with transmitting data due to the level of data being received. The increased rate of decrease in user time available also shows what can be a significant effect on the non-IO related elements of the jgViz client node (such as those calculating the moving short and long term averages). However, figure 6.21 shows that the performance of the graphics pipeline itself is not massively affected. Indeed, the seemingly unrelated nature of the client and the running pipeline suggests that this is a combination of two things. Firstly, that we may be seeing effects of network switching equipment saturation (a Gigabit switch may or may not be able to simultaneously switch multiple channels at full rate). Secondly, that the pipeline member nodes may be slowed slightly by a very small increase in the time it takes them to get an answer to a query back to the monitoring jgViz client due to the network saturation on the client node. The network statistics for the client (figures 6.24 and 6.25) show the relative differences between the congested and non-congested situations. In terms of the transmitted data, the increase seen is due to TCP acknowledgements having to be sent. However, the proportional increase here is viewable and the amount of transmitted data is still not so much as to significantly contribute. The received data shows the network saturation that has occurred and that this is very much greater than in the non-congested state. In both graphs, we see a clear fall-off at the 8-node mark before 'recovery' back to the high-level at the 16-node mark. This appears to be related to the processor utilization seen in figure 6.23 and the major transfer from

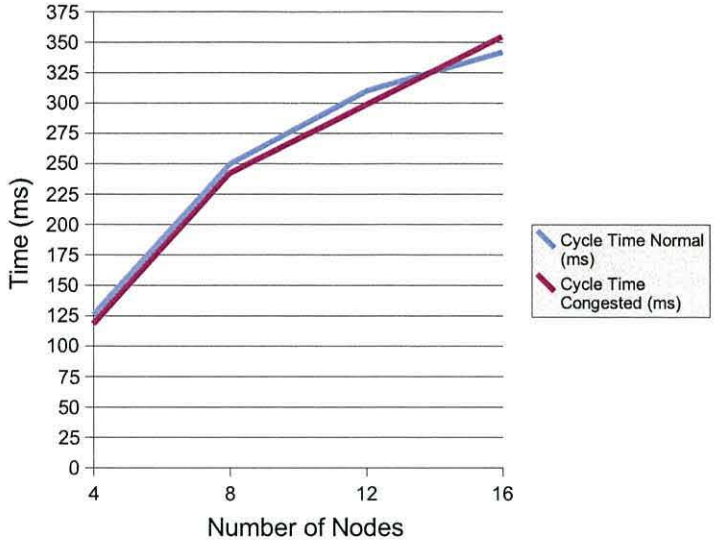


Figure 6.26: Client Monitoring Cycle Time in Gigabit Tiled Pipeline with Congested Application Node

idle-time to system-time seen at this point. We would hypothesize that this is caused by some operating system behaviour related to the network stack - perhaps the 'collision' factor between transmit and receive reaches a certain level.

Congesting the Application

The application node in a jgViz running pipeline will only normally be transmitting data to render slaves. Depending on the processing requirement of the application, the complexity and volume (in both geometry and texture data) of the graphics output and the splitting of the graphics that has to be performed, the application node's processing elements are likely to be the busiest in the entire pipeline. As the performance of the application and its graphics element is the defining quality of pipeline performance, the effect of network congestion at the application node is particularly relevant to the entire jgViz experience.

Figure 6.26 shows that network congestion at the application node does nothing to affect the monitoring cycle time of the client. The application node is still able to answer network requests (i.e. of its GRIS LDAP server for performance data) sufficiently quickly that no noticeable difference is made to the client. However, figure 6.28 shows how the more intense task of rescheduling a pipeline is affected more, if still not tremendously significantly. This will be caused by contention for the network into the application node between the

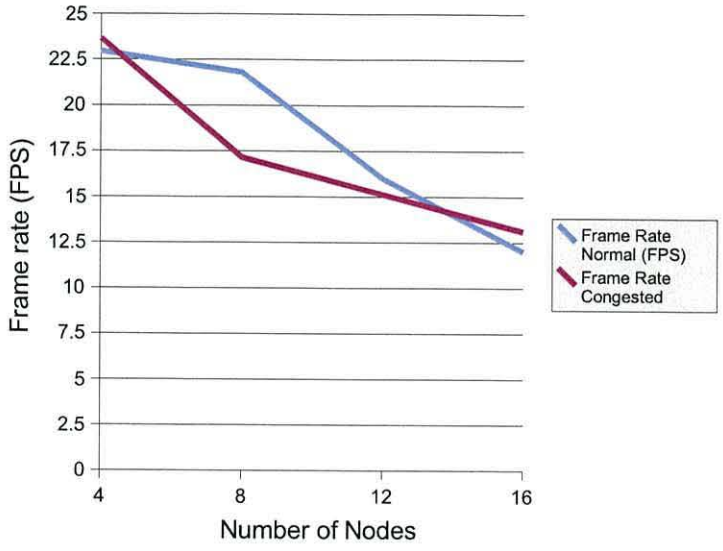


Figure 6.27: Visualization Frame Rate in Gigabit Tiled Pipeline with Congested Application Node

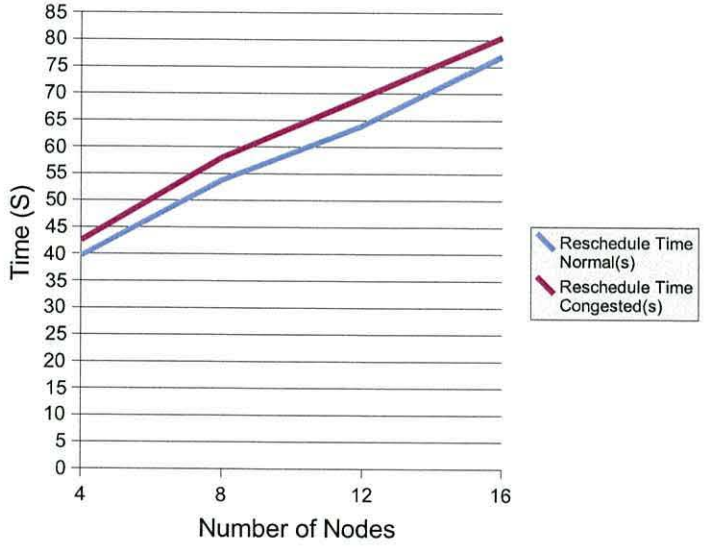


Figure 6.28: Reschedule Time for a Gigabit Tiled Pipeline with Congested Application Node

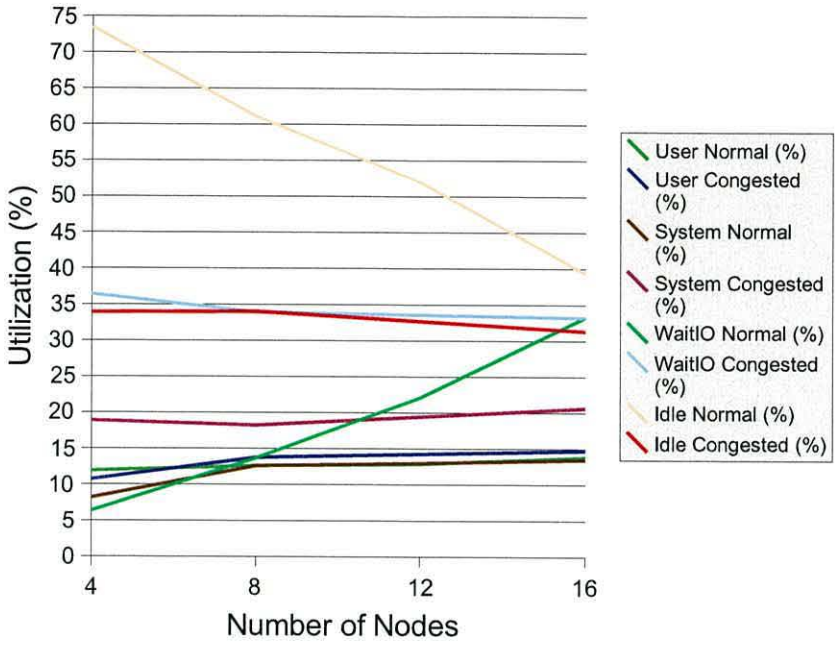


Figure 6.29: Client CPU Statistics in Gigabit Tiled Pipeline with Congested Application Node

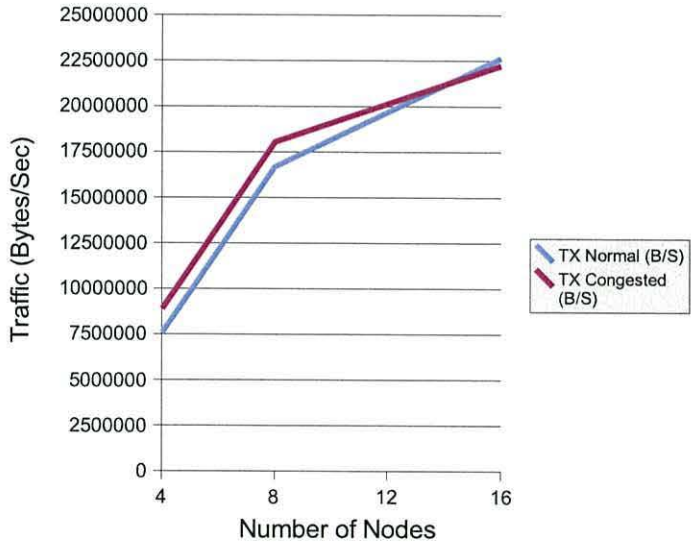


Figure 6.30: Client Transmitted Network Traffic in Gigabit Tiled Pipeline with Congested Application Node

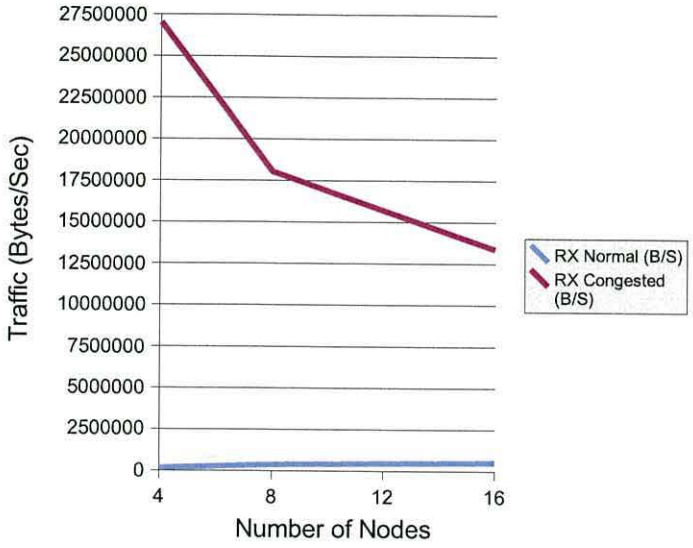


Figure 6.31: Client Received Network Traffic in Gigabit Tiled Pipeline with Congested Application Node

client trying to stop and start jobs, and the network congestion cause. Regarding pipeline performance, Figure 6.27 shows the varied impact of the congestion. Here, network congestion introduces a definite unknown and unpredictable element. In comparison to the same experimental results for the congestion of the client machine (figures 6.20, 6.21 and 6.22) we see the expected result of the application node congestion having more effect on the running pipeline (due to the fact that the application node is a fundamental part of the pipeline whereas the client is merely an 'on-looker'). Figure 6.29 shows the change from the non-congested situation of idle time transferring to wait IO time as the number of nodes increases, to the congested situation of these two statistics being very similar. There is also increased system time in the congested application node case, again due to the interrupt servicing for the received network traffic. The changing balance of received vs transmitted network data (see figures 6.30 and 6.31) in the congested network is reflected as delivering consistent system and wait IO CPU statistics whilst the user time is very similar to the non-congested results. Observing this changing network balance, figure 6.30 shows us that the congestion of the network input has very little effect on the network output from the application node to the render slaves. Although the network connection is running full-duplex, figure 6.31 shows that the received traffic on the congested application node reduces as the number of nodes increases. This is believed to be due to saturation of the internal network capabilities of this node, whether due to bus or network driver/chipset limitations. Even as part of a 16-node pipeline, the application node still manages to receive over 100Mb/sec

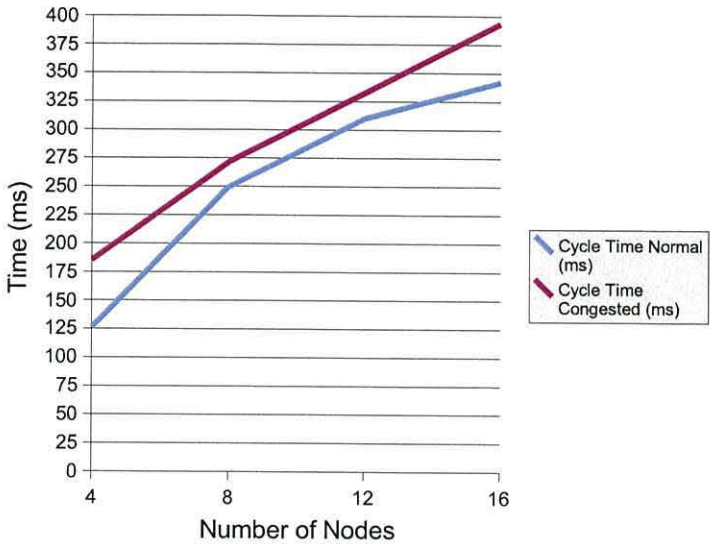


Figure 6.32: Client Monitoring Cycle Time in Gigabit Tiled Pipeline with Congested Render Slave Node

of traffic, more than the Fast Ethernet ever did!

Congesting the Slaves

Comparing the results for the congested slave network to the congested application node network produces its own intrigue. Figures 6.32 and 6.34 show the expected generally slightly worse performance with poor performance at the low number of nodes in the monitoring cycle measurements and good performance at the same number of nodes in the reschedule time. Investigations to diagnose this minor difference have been inconclusive. Figure 6.33 shows the total effect on the entire pipeline performance to be decreasingly susceptible to a single node being on a converged network. This makes sense as the overall frame rate drops and the requirements of an individual render node lower. The far larger reduction in performance at low node counts certainly indicates that the congested network affecting a higher relative proportion of the render nodes has a serious effect. This is the situation that jgViz is designed to prevent before it becomes a serious issue. Figure 6.35 indicates the same order of wait IO impact resulting from the network congestion as was the case with the application node - slightly more in this case as there are less other tasks to keep the machine busy on something else. Again, it is also seen that this increase in wait IO is directly related to the reduction in idle time while user time exhibits very little change. System time increases in line with the network traffic also being received, although

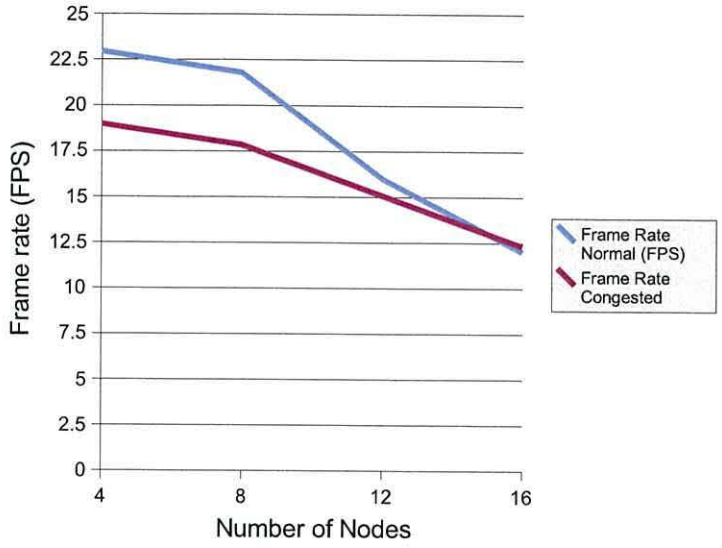


Figure 6.33: Visualization Frame Rate in Gigabit Tiled Pipeline with Congested Render Slave Node

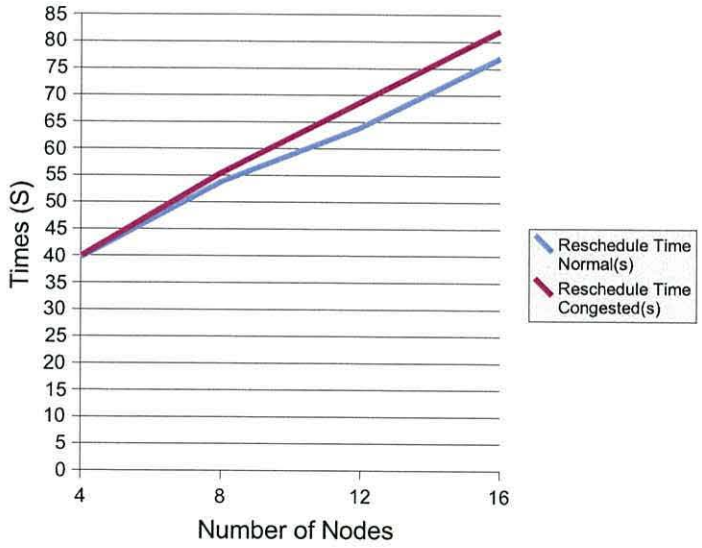


Figure 6.34: Reschedule Time for a Gigabit Tiled Pipeline with Congested Render Slave Node

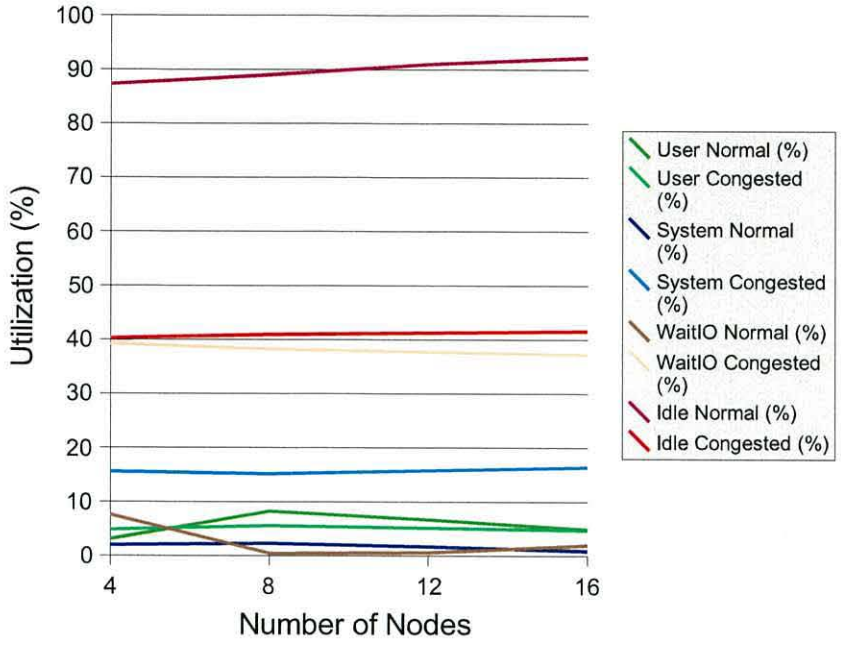


Figure 6.35: Client CPU Statistics in Gigabit Tiled Pipeline with Congested Render Slave Node

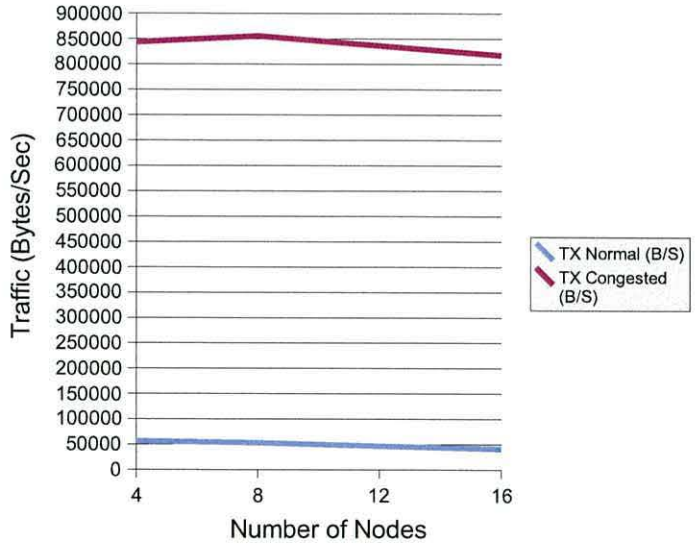


Figure 6.36: Client Transmitted Network Traffic in Gigabit Tiled Pipeline with Congested Render Slave Node

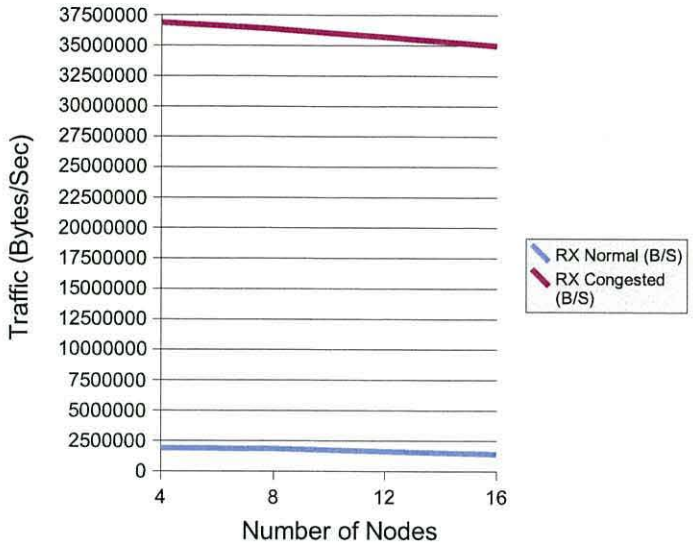


Figure 6.37: Client Received Network Traffic in Gigabit Tiled Pipeline with Congested Render Slave Node

these results generally show that it is both system and wait IO time that increase due to the converged network. It is safe to assume that the wait IO increase is due to the interruption in the graphics stream caused by the additional network traffic having to be acknowledged. As a render slave does not have transmit responsibilities in a tiled pipeline, the traffic levels seen in Figure 6.36 confirm our previous postulation for the application node being congested by the level of traffic generated in simply acknowledging the TCP segments received. We know that this level of traffic, whilst relevant, does not significantly affect the performance of a node. The data level received by the render slave (Figure 6.37) is seen to decrease slightly as the node count increases, even though this machine is the target of the network congestion. The larger node count means that the application node is producing less traffic per slave and that the slaves are seeing a longer period of receiving data for a single frame as the application node works harder. This causes a small rise in the interrupts that the slave's processor must service (as seen in Figure 6.35) and a small overall traffic reduction.

6.5 Optimizing jgViz's Monitoring and Rescheduling

Reliability and predictability of the running graphics pipelines are both managed by jgViz. The monitoring and rescheduling functions of jgViz are linked by the decision-maker, which

is responsible for deciding when the performance statistics gained through monitoring justify triggering a reschedule. As previously discussed, this is critically dependent on a number of user settings - size of long and short histories and the proportion of change between them required to trigger an 'event'.

These settings are easily adjusted by the user, but need to be optimized as a default and for the purposes of proving how well jgViz performs. Different technologies will require changed values as the network and client node speeds increase and change the balance of timings in the monitoring stage. This will depend more on the latency of such technologies than the bandwidth they afford so Fast and Gigabit Ethernet are both going to exhibit similar trends within their performance limits.

The process of optimizing is largely empirical and by learning through experience. The principal difficulties are as follows:

1. The 'long-term history length' values (both latency and utilization) cannot be too large as jgViz cannot begin comparing the short and long averages for either metrics until the long-term buffers are full.
2. The 'short-term history length' values (both metrics) need to be long enough to overcome brief, transient factors affecting the metrics.
3. The short and long history lengths need to be far enough apart to be representative, and differ sufficiently.
4. The 'proportion change required' parameters need to be set in concert with the history length settings and should not be so low as to trigger a needless reschedule but should not be so high as to not trigger when required.

The process of optimizing these values took several steps and we began with the following values for our parameters, as simply a best initial guess:

- Short-History Load Length = 30
- Long-History Load Length = 180
- Short-History Net Length = 30
- Long-History Net Length = 180

- Load Proportion Change Required = 80
- Net Proportion Change Required = 80

Experimenting with these settings in a relatively small setup (4-nodes, tiled on a single LAN) showed that jgViz was too likely to reschedule after a fairly short duration, making the pipeline somewhat 'brittle'. Observing the full debugging output from the jgViz monitoring subsystem, we observed that this was primarily due to subtle variations in the network latency that was quickly becoming a difference of greater than 80% in short to long average comparison. Reconsidering this, it seemed logical that increasing the required proportion change would be the best solution. Our initial investigation had not adequately considered the case of the 'normal' loading on the slave nodes (under monitoring) being low such that a small value change represents a great proportional change. We also observed, however, that the small latencies of a network within a LAN (sub 1 millisecond) are very much more susceptible to slight "bumps" here-and-there than load data from an operating system is, due to the software layer of abstraction involved in reporting load data. Hence the following increases to the proportion changes:

- Load Proportion Change Required = 200
- Net Proportion Change Required = 400

This gives a more balanced control of the pipeline. There are no unnecessary reschedules due to normal network intricacies. However attempting to cause a reschedule by remotely logging into one of the slave nodes and running a compute intensive task works, but takes too long to trigger (in excess of 30 seconds). The solution to this is to shorten the short and long buffer lengths. The short-term buffer length is key, but it makes sense to also reduce the long-term buffer length if we can, in order to save memory (less significant) and compute resource (slightly more significant) on the jgViz client performing the monitoring. The changed values are:

- Short-History Load Length = 10
- Long-History Load Length = 60
- Short-History Net Length = 10
- Long-History Net Length = 60

These are the final values that we found optimized the monitoring and rescheduling process. Further adjustment of these values for individual setups can be made, but it is likely that the smallest of changes to the surrounding network environment would mean they required updating again. Given the relatively variable quantities being dealt with in different applications and even in running things at different times, optimizing to such a level seems pointless. Scalability to large numbers of nodes (beyond 32) will be limited by the capabilities of the jgViz client machine as the length of the montioring cycle time (time to gets updates from all nodes) may become ineffectively large. In such a case, the above settings could be optimized to reduce the downside of running the jgViz client on a less capable machine than would be ideal.

6.6 Conclusions

In commenting on the experimental results we have seen, there are (as we would expect) definite patterns. Generally speaking, tiled pipeline performance is decent whereas readback performance is unacceptable. This is primarily due to the 'double-hit' of network latency suffered in a readback pipeline. The length of the pipeline (in stages rather than cable length) is the overwhelming factor. This ties in with considering the latency of Ethernet as a network for distributed visualization. We find that Fast Ethernet is unsuitable for all but the smallest and least-demanding of uses as it suffers the limitations of both latency and bandwidth and simply cannot transport sufficient data around quickly enough. This is in contrast to Gigabit Ethernet, which provides genuinely usable performance by exhibiting shorter latency and enhanced bandwidth. We also see that readback pipelines in particular suffer depending on the rendering hardware used. Reading pixels from a frame buffer is a task for which there has been little optimization in past graphics rendering solutions where the focus was to get pixels to the screen as fast as possible, not from it. Software rendering and read-back imposes an additional constraint as hardware rendering has only really received any optimization for the task in recent graphics processing units on the new PCI-express interface. It is worth noting that the bandwidth limitations we observed with the Gigabit Ethernet may be limited due to the hardware/software combination we have. We have in the past seen three times the bandwidth achieved using a different Gigabit network card on a less powerful system, but with official (vendor supplied) network drivers. The particular hardware and driver combination used here may be the source of the limitation.

Grimstead *et al.* have carried out a performance study of the RAVE system [65], although

it is difficult to draw similarities or comparisons with jgViz due to the different abilities and aims of the two projects. RAVE's utilisation of Java and Java3D places an additional limitation on performance in exchange for code that does not need to be recompiled for different platforms. However, the comparatively limited degree of parallelisation, the lower resolution, the set nature of the visualizations, and the distinct difference between local and remote rendering in RAVE, all lead to different characteristics to jgViz. The concept of a heuristic of some kind to determine the utilization of a GPU is something that could be used as a jgViz metric as well as within RAVE and other Grid visualization projects.

The effects of network congestion are clear in showing that the combined effect of additional traffic, the increased CPU utilization needed to service it and the increased contention for resources creates a problem. The more significant effect of limitation is not on pipeline performance, but on jgViz monitoring and rescheduling performance. Although the monitoring cycle time does not suffer too much (as it is such a lightweight request to service for the pipeline nodes) the sometimes substantial increase in the reschedule time can be significant. If a node becomes the subject of network congestion, but it is at such a level as to pass 'under the radar' as far a jgViz's network monitoring is concerned, then we may have the problem of jgViz not rescheduling when it should. Within the current metrics only optimization can be performed, and it is difficult to see how else such a problem could be detected without performing fake reschedules repeatedly.

Of particular interest is the fact that both the distributed graphics pipeline elements and the jgViz system produce a broad spectrum of requirements from hardware due to the balance of IO and compute work. The implication of this is that improving performance requires an all-round approach. However, the degree of CPU time spent in the *wait IO* state indicates that a different network technology would be the most useful thing to look at initially, perhaps Myrinet or Infiniband with the low latencies both technologies offer. Whether we then encompass Cluster computing is a worthy question, however, as studying non-optimal equipment is our concern. The application node is unsurprisingly seen to be the most loaded node and so work on optimizing the selection of such a node as part of the jgViz scheduling stage is important - for example in picking a SMP system so that a single processor does not have to split its time between the application generating graphics and the Chromium element partitioning that graphics stream and sending it out to render slaves. Further work going on in the Chromium community on multi-threading this process is particularly interesting as it will enable an immensely complex graphics application to be run on a compute-optimized, expensive Grid-attached machine whilst the rendering is carried out by graphics-optimized, cheap PCs that are also Grid-attached. The heterogeneous Grid really

comes alive when such systems can be implemented.

Improving the level of knowledge we have about the application we are intending to run and, alongside, increasing the intelligence in the scheduling element of jgViz to suit the wide-variety of hardware available is another area for improvement. Characterisation at the instruction level (perhaps using a technology such as DTrace [87]) is worthy of study.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Visualization is a process that can enable the understanding and interpretation of datasets that a human may otherwise find difficult to comprehend. High-performance visualization represents taking visualization into a high-performance computing world of parallelism and supercomputing-class resources. Grid middlewares are collections of software components aimed at integrating services and resources, and dealing with the security of and access to such resources. Beyond those terms, the definition of what Grids are varies, but they are often noted as being (in some form) the future of the connected world. Whilst there has been work on visualization within Grids, it has mainly been focused on relatively specialized applications. In this thesis, however, our target audience is researchers and developers wishing to use existing visualization applications based on standard graphics interfaces and enable them to run in Grid environments. We have investigated the feasibility of achieving this goal and so providing high-performance through parallelism and improvements to the visualization experience (perhaps generating a higher-resolution image, for example).

Chapter 2 of this thesis presented an overview of visualization and Grid technologies before moving on to discuss those projects that have attempted to combine these two areas. We noted that the Chromium project has been widely used and is well regarded. We discussed the implications of the recent generation of commodity graphics processors and ever higher performance networks, and concluded that the era of supercomputer visualization may be under threat. The connectivity of the Grid is seen as the architecture of the future for large scientific studies. However, in speculating about a Grid visualization system, we

identified a need to study existing interactive visualizations in a non-optimal, commodity hardware environment and selected Chromium as our vehicle for achieving this within the Grid. Chromium, however, has not been implemented with the Grid in mind, and so much work was needed to do this.

Chapter 3 proposed an *information model* for *jgViz*, our Grid visualization project. We developed a specification language based on the necessary configuration data to configure a multi-node Chromium graphics pipeline and added data as needed in order to discover, schedule and launch such a pipeline as a Grid-enabled application. We split our model into a Grid server component that advertises the capabilities of a node, and a client component that is part of the client application. We developed a simple path for the information through the use of the Grid Information Service (GIS) standard. The hierarchical nature of this standard enables the server to advertise capabilities in a Grid Resource Information Server (GRIS) which is a subordinate of a Grid Index Information Server (GIIS), that is searched by the jgViz client for single-point resource discovery. We utilised a feature of the LDAP language and developed a customized schema in order to represent our data in an easily-searchable manner. We believe our language does not unnecessarily complicate the job of describing a pipeline, which is important as a system administrator would have to configure some language elements manually. We also believe that our information model makes good use of the GIS standard and sits well with its principals of dynamism, security, federation and searchability.

In chapter 4, we discussed the scheduling process that we have developed for utilization of Grid resources. We selected a two-stage scheduling process involving a first subsetting on the basis of all discovered nodes that offer suitable configurations and a second subsetting on a ranked scoring basis. The two simple metrics of proportional load and network latency proved to be a good basis for establishing nodes scores due to their lightweight nature and the fact that we do not wish to detrimentally affect any discovered nodes. For the purposes of handling tiled display pipelines, the scheduling process utilizes necessary added data in the information model and, for scoring purposes, considers a tiled display by simply calculating the mathematical mean of all nodes in the display. Competing tiled displays can then be compared fairly. We made a conscious decision to make the jgViz client application as component-based as possible, and so the scheduler's output is only in the form of an augmented Chromium configuration script that can be utilized separately, having taken advantage of the jgViz system to establish and create the configuration.

Chapter 5 discussed the use of jgViz for launching and controlling a distributed graphics pipeline. We carried out a brief study of the problems associated with 'ownership' of

graphics hardware and workarounds for this, but we identified that a fuller solution is needed for remote access to graphics processors. In launching a pipeline, we utilise several Grid protocols. GridFTP and Global Access to Secondary Storage (GASS) are used for data transfer, although [61] indicates that they do not offer the necessary throughput performance for live graphics data. The Grid Resource Allocation Management (GRAM) protocol is used for launching and stopping the jobs on Grid nodes for the entire graphics pipeline to operate. We developed a monitoring system that 'watches' a running pipeline for performance problems utilizing the same data metrics as originally used to schedule the nodes in the pipeline. As a running pipeline requires temporary 'possession' of the nodes involved, the 'aggressive' concurrent monitoring of all nodes is used here. This compares to a less rigid round-robin poll through all the nodes monitoring utilized during the initial scheduling. In developing a means to decide when a pipeline's performance has become unacceptable, we found that the metrics were overly sensitive to very short-lived spikes in the metric data. Hence, we have utilized a system of two rolling averages - of short and long time length - for each metric, and our monitor process considers the pipeline to be unusable when the difference between these averages for a single metric exceeds a set proportion. Should an event be triggered, then we have implemented a system to stop the running pipeline, remeasure and reschedule amongst the available nodes, and then start a new pipeline. This is subject to a time delay, however. We have implemented a mechanism to preserve the application state during a reschedule, through the simple recovery of a state file. We speculated on a number of additional or alternative metrics that could be used, including the possibility of code characterisation and the use of additional Grid services during both the initial scheduling and the runtime monitoring. Our pipeline launch and monitoring system makes effective use of Grid standards and offers a service currently focused on intra-organization use, as a few issues remain to be addressed for true inter-organization use, principally regarding Grid protocol based secure and high-performance streaming of visualization data between nodes.

In our extensive study of the performance of distributed graphics pipelines (chapter 6), we have shown that a non-purpose built hardware setup can be utilised for distributed graphics visualization on the Grid. There are certain limitations to this, with the principal issue being that Fast Ethernet is broadly not good enough as the interconnect in a parallel graphics pipeline. Readback performance also suffers greatly depending on the hardware available and so is likely to become an area of more interest as some of the specific issues involved are solved through future technology development. Gigabit Ethernet has shown more than adequate performance for tiled configurations and actually shows some speed-up when increasing the numbers of render nodes in the pipeline, within certain bounds. By

flooding individual nodes in a running pipeline with network traffic, we have concluded that the pipeline is subject to congestion on individual nodes, but modern PC systems are fast enough for the interrupts generated to not interfere too much. Further, such a network load in normal use is unlikely. We have also optimized the parameters of the monitoring and rescheduling element of jgViz and we find that this capability performs well, although it is subject to limitations in timing by virtue of what it tries to do. There is some inherent delay in a pipeline being rescheduled that we believe becomes prohibitive when larger numbers of nodes are involved, and this remains an issue.

With regard to the hypothesis originally stated in chapter 1, we are able to conclude that a general-purpose, real-time Graphics pipeline can be provisioned using Grid technologies within the limitations as set out above. We also see great prospects for improvement and further work, as laid out in the next section.

7.2 Future Work

Researching in such a topical, diverse and widespread area as Grid computing has a good side and a bad side. It is good to be part of something that is currently in vogue and which has a bit of momentum. It is bad (or, at least, difficult) to ever develop a perspective view of such a breadth of work. The work presented in this thesis, however, has led to several ideas for future development and research related to the particular field of Grid visualization. We summarise these below.

- The recently completed Web Services Reference Framework (WSRF) [40] Grid standard and the corresponding Globus 4 implementation likely represent the future of Grid middleware, in concept if not instance. We observe with interest the development from the reasonably well accepted Globus 2 through Globus 3 and the Open Grid Services Architecture (OGSA) to this latest generation. We also note that whilst a change of thought process is somewhat involved in implementing new style Grid software, the designers have paid attention to existing Grid applications and developers. With specific regard to the information model jgViz component, OGSI or WSRF Grid software, such as the *GeneGrid* [88] and e-Viz projects [89], is required to implement custom service registries to contain resource status data. Although the jgViz information model currently utilizes the MDS, we do not believe there would be substantial difficulty in moving from such an LDAP based approach to an OGSI/WSRF web service based approach - the difference is in style, not content. We believe,

therefore, that a change in jgViz to using the WSRF would be a worthy, if significant, piece of future work as part of jgViz's development.

- The jgViz system should be enhanced to include support for automating the deployment of the mothership and application nodes as part of the Chromium pipeline. Given that the application node in particular may need to be carefully located, the user may wish to interact with this in some way.

- As mentioned at several points within the thesis, further experimentation and study is required in relation to additional metrics being used during both the scheduling and monitoring stages of the jgViz client. The possibilities in areas such as code characterisation and advance-reservation are definitely worthy of investigation with regard to improving the initial scheduling. For the monitoring stage, it is somewhat more challenging. Consideration has been given to using the frame rate achieved by a visualization task as a measure of its effectiveness, as this could easily be obtained from the Chromium system. However, further user interaction is then required to establish the range of frame rates that were effective. A rolling averages approach could be deployed here as it is with the two existing metrics, however, the reschedule process for a stopped pipeline would not be guaranteed to detect any change in the available nodes and so would be more likely to reschedule the same pipeline. A more enticing possibility for use at both the initial scheduling and the monitoring stages is the use of Grid-monitoring services such as the Network Weather Service (NWS) for prediction and monitoring of network traffic levels. Careful study would be needed to ensure that the impact of such services would not be too detrimental to running pipelines, but integration of such a scheme alongside an advance-reservation system is certainly attractive. We believe that such systems are likely to become an increasing part of Grid applications and deployments, and would fit in well with a transition of jgViz to the WSRF-based 'next-generation' Grid. The idea of some sort of 'GPU metric' mentioned in Chapter 5 is also enticing. Another issue with all deployment that is considered is the resource cost of people - we have observed that additional systems administrator time in order to implement and support Grid systems is difficult to fund - so, we need to minimize such requirements in our implementation.

- In parallel with improving the metrics used in scheduling and monitoring machines, improvement of the monitoring hierarchy is worthy of research. Whilst the existing information model works well, the implementation of a more hierarchical system of monitoring (using Grid standard protocols) would ease the transition to a larger number of nodes by distributing some of the connection load currently placed on the single client machine. Whilst this would help scalability during both initial scheduling

and monitoring, it would need to be carefully controlled so as not to become too disruptive to running systems and pipelines.

- The time duration for jgViz to reschedule an under-performing pipeline requires attention. Whilst it is tolerable with small node count pipelines, the time taken is unacceptable with larger node counts. This interferes with what is supposed to be an interactive, real-time experience. However, it is the nature of the Grid protocols that a time delay is encountered. We have considered the possibility of performing the startup of the new pipeline whilst the old one is still running in order to reduce transition time. However, this does not fit well with graphics processor ownership/access issues, or the situation where a node is involved in both the old and new pipelines. More work is needed on this, perhaps involving some implementation of two stages in the startup of Chromium components such that these issues can be bypassed.

- Security needs dictate that Grid protocol transport of inter-node visualization data is an objective. Whilst Grid data protocols do not yet offer the necessary performance, future hardware and protocol improvement will make it possible to move data in such a way. The current Chromium native transport can use any network port and so can be placed onto one of the Globus ephemeral ports. However, the security aspect of tunnelling this data through a secured Grid transport is highly desirable for inter-institution work in a Grid. This is increasingly relevant as the multi-institutional element of the Grid, involving current and future high-performance national and international networks, leads to greater resource sharing within Grid virtual organizations.

- The initial work presented on the transfer of state between generations of a rescheduled pipeline could harness expertise in other fault-tolerant research, be it from other Grid work involving Grid standards or from, for example, peer-to-peer or cluster computing. Fitting any low-level system into a Grid hierarchy is going to be difficult due to the heterogeneity and diversity of Grid components.

- Planned improvements to the Chromium software will also improve jgViz's performance. Specifically, multi-threading of the 'tilesort' Stream Processing Unit will enable this processor-intensive task to be carried out on a high-performance multiprocessor machine as the application node. This will greatly increase the scalability of a Chromium pipeline to deal with very sophisticated graphics and fits in well with the vast development in graphics processor performance on the rendering slaves. The possibility of using such extra processing power to assist with data transfer over the network, for example through stream parallelism, is also exciting. jgViz would then crossover very effectively between the differing machine classes within a Grid in uti-

lizing each for the task they are best suited. This is a particularly interesting area for future work.

- Two other items are on the wish list: further experimentation with differing hardware, particularly in using hardware native rendering and higher performance networks, as nodes and clusters utilizing such technology are as likely as anything to be attached to a Grid; and implementation of a user adaptable graphical representation of a constructed graphics pipeline within the jgViz client.

All in all, we believe the design of jgViz provides a sound basis for future development. It seems likely that a good proportion of this work will involve greater integration with other Grid projects as the world moves forward. However, there is no point in jgViz becoming just another Grid visualization project so it must be focused on flexibility, and catering for the same generic graphics applications as now. We also believe that with future development as discussed above, jgViz would only become more useful.

Appendix A

Ethernet Effect

The results observed in chapter 6 regarding performance of the two network standards we have practical experience with are worthy of further discussion, as much for explaining things to ourselves as anything! Table A.1 recaps these results.

Bandwidth

Initially looking at the bandwidth figures, we see the limitation of 269Mb/Sec. This number was shown up repeatedly with tests on different equipment and in different forms. We performed the same tests in 3 environments and got the same basic number:

1. Between two of the test machines over the 50 meter cable runs to the Cisco Catalyst 4006.
2. Between two of the test machines over a 1 meter cable run to a Netgear 5-port mini Gigabit switch.
3. Between two of the test machines over a crossover cat5e network cable.

Network Technology (Mb/Sec)	Real-World Speed (Mb/Sec)	Latency (us)
Fast 100	93.93	124
Gigabit 1000	269.00	93

Table A.1: *netperf* and *ping* measured real-world network data transfer rate

As we have tested different switches and cables, it seems fair to assume that the limitation here is with the network subsystem of the test machines. Or, could there be a fundamental limitation in Gigabit Ethernet that we are bumping into? We have borrowed two other machines to test this theory.

Figure A.1: Second Gigabit test rig

Figure A.1 shows our Gigabit test rig. We use two Sun Fire V20Z servers (with dual 2.6GHz Opterons, 4GB memory, Broadcom Gigabit Ethernet) and a Cisco Catalyst 4006 (with 24 Gigabit ports). We performed both TCP and UDP tests with this equipment and also recorded CPU utilization. Table A.2 shows these results in comparison with those from the test machines we have.

Equipment	Test	Bandwidth (Mb/Sec)	Transmit CPU (%)	Receive CPU (%)
Athlon 3.2GHz PCs	TCP	273.26	41.96	51.42
	UDP	289.9 (0.2% loss)	37.03	29.56
Sun Fire V20Zs	TCP	907.32	7.55	12.81
	UDP	1098.63 (15.9% loss)	22.85	10.05

Table A.2: Gigabit Ethernet testing results

These results prove that there is no implicit Gigabit Ethernet limit, no limit associated with our cabling and no limit associated with the use of the switches we have available, the Catalyst 4006. The limitation must be in the hardware/software combination in the Athlon PCs. The more expensive Sun server solution provides stunning performance, showing the advantage of quality hardware and a commercial vendor's software driving it. The CPU utilization also shows much lower utilization by the Sun servers, in spite of the fact that are handling nearly four times the traffic. The degree of packet loss experienced by UDP on the Sun servers of 15.9%, reflects the network technology becoming a limitation, hence we do not suffer as such at the lower 290Mb/Sec level of the Athlon PCs.

Latency

Our results show that Gigabit Ethernet has a lower latency than Fast Ethernet. Can we explain the reason for this?

First, a bit of theory. Fast Ethernet in this case is actually the 100-BASE-TX variant of IEEE 802.3u. This takes advantage of the 125MHz capability of category 5 cabling to only require 2 twisted-pairs (of the four available) and uses an encoding scheme that includes 2 signalling levels. The encoding scheme effectively gives 4 bits of data for every 5 clock cycles (explaining why transfer is 100Mb/Sec whilst the clock rate is 125MHz). Gigabit Ethernet here is the 1000-BASE-T variant of the IEEE 802.3z standard. This standard uses all 4 twisted-pairs in a category 5 cable. It uses a different encoding scheme that includes 5 voltage levels, representing 00, 01, 10, 11 and a special control signal. By sending a voltage level down each twisted-pair, 8 bits can therefore be sent in parallel per clock cycle. This is still running at the 125MHz category 5 capability, but transferring ten times the data.

An ICMP Ping is defaulted to be 56-bytes in length, although there is also the 8-byte IP overhead added to that to produce a 64-byte (512-bit) data transfer as necessary for Ping to transit an IP network. Fast Ethernet sends (almost) one bit per clock and so takes eight times the time of Gigabit Ethernet which is sending 8 bits per clock.

We can bump up the size of the Ping packet to try and make the Gigabit Ethernet take as long as the Fast Ethernet. The factor is, once again, 8. So, the Ping packet size becomes (8 * 64)=512 bytes, but we need to deduct the 8-byte IP header, making the Ping payload 504 bytes. The Maximum Transmission Unit (MTU) of Ethernet is 1500 bytes, so this data quantity still fits within an Ethernet frame. Table A.3 shows the results of this alongside other latency data.

Test Details	Latency (us)
100-BASE-TX 50m	129
100-BASE-TX 1m Crossover	130
1000-BASE-T 50m	96
1000-BASE-T 1m Crossover	95
1000-BASE-T 50m Large Payload	126

Table A.3: Latency testing results

Although not an exact match, these results do show:

- That cable length makes no difference
- That the presence of an unloaded switch makes no difference
- How increasing the ICMP payload over Gigabit increases the latency in close proximity to what we would expect
- The dependence of latency on frame size

As such, this answers the posed question. It is important to remember that any traffic is far more than just a single bit of data.

Appendix B

The jgViz Client GUI - an Implementation Overview

The overwhelming majority of code for this project is finalised in the jgViz client GUI. This is a Java2 application that makes use of various libraries to interact with LDAP and Grid resources. Java was chosen for two primary reasons. Firstly, other languages are hard pressed to match Java's cross-platform abilities; secondly, Java's Swing API makes writing GUIs easy. Most of the code that is in the jgViz client is the result of many different testing programs written to test out new parts with fixed input data. Due to jgViz's nature as a research project, the code structure and design has been pieced together with ideas expanding upon ideas repeatedly. It is, therefore, not an optimal implementation as reflected somewhat by its class diagram, shown in figure B.1.

The core structure of the code reflects the manner in which the various components of the client were developed - one major component in the process of using a jgViz pipeline at a time. The ordering here was mothership, application node, scheduling, configuration script, run and monitoring. These became the fundamental foundation of the code by becoming different 'panels' of the GUI. The user simply passes through the tabs from left-to-right to setup a pipeline. The contents of each of these panels is relatively obvious:

Mothership Panel Contains settings relating to the Chromium mothership to be launched - where it is, path to Chromium on that machine, etc.

Application Node Panel Contains settings relating to the Chromium application node

Scheduling Panel Contains pipeline related settings

Configuration Panel Generates and contains the Chromium configuration script

Run Panel Takes the configuration script and launches the pipeline components

Monitoring Panel Monitors the running pipeline components and reacts accordingly

There are a few key data structures that hold data within the jgViz client. These are instantiated in the main *jgvFrame* class and passed to those frame objects created (such as the *jgvApplicationNodePanel* for the application node) that need them. Discovered resources are stored in a vector called *availableNodes* that is loaded with data by the resource discovery code (also in the jgvFrame class) and used by the scheduling code. The output of the scheduling process is another vector, this time called *activeNode*, that is used to generate the configuration script. The configured pipeline is then launched by the run panel extracting the relevant data from the passed configuration script and the monitoring panel learns the nodes it needs through vectors shared with the run panel. Custom classes are implemented for holding the data for each type of node and the various pieces of user-configurable settings. These are simply held in the above vectors as objects and cast to get the data back out when needed. The vectors effectively act as temporary storage locations for data on nodes.

The resource discovery process was the first part of jgViz to be implemented and is part of the *jgvFrame* class. This makes use of the Java class libraries originally developed by Novell but now part of the OpenLDAP project. Searching of the LDAP database is achieved through the use of the jgViz LDAP schema that provides an *objectClass* for all jgViz nodes advertised. This eases the process of searching for and discovering such records as the LDAP search functionality allows us to restrict on the basis of *objectClass*. Once available nodes are discovered, the scheduling panel component also uses the LDAP libraries to get updated metric information for the nodes and also launches the ICMP Ping metrics gathering shell script.

The run panel launches the pipeline through the Java CoG toolkit as previously described and keeps the standard output and error streams from the launched processes in a vector. The output of these streams is then routed to the three frames in this panel. The vectors containing the launched jobs and all other maintained references to them are passed to the monitoring tab when it is selected and self-populates. This makes use of the same LDAP and shell methods to update the two metrics for each running node. However, here they are performed concurrently. In the scheduling panel, a thread is spawned to run alongside

the GUI (and hence keep it up-to-date) to performing the actual round-robin cycle through the discovered nodes updating metric data. In the monitoring panel, in order to provide concurrent metric update, a thread is once again spawned alongside the GUI to update the active nodes, but that thread then goes on to launch a separate thread for each node to be updated. So, lots of thread are launched and then spend time waiting for I/O before returning. Only once they are all returned does the cycle go round again. As we have seen, however, this doesn't tend to take long.

Should the monitoring stage detect a problem with the pipeline and trigger an event, then it uses references to the other panels that it has to call the various methods within those panels that are necessary to reschedule the pipeline including the process of trying to save a state file. This process therefore involves: the stop method from the run panel, use of the Grid toolkit, calling of the scheduling panel's methods to update and reschedule the nodes in the pipeline, calling of the configuration panel's configuration script rebuild method and finally calling of the appropriate methods within the run panel to start the new pipeline.

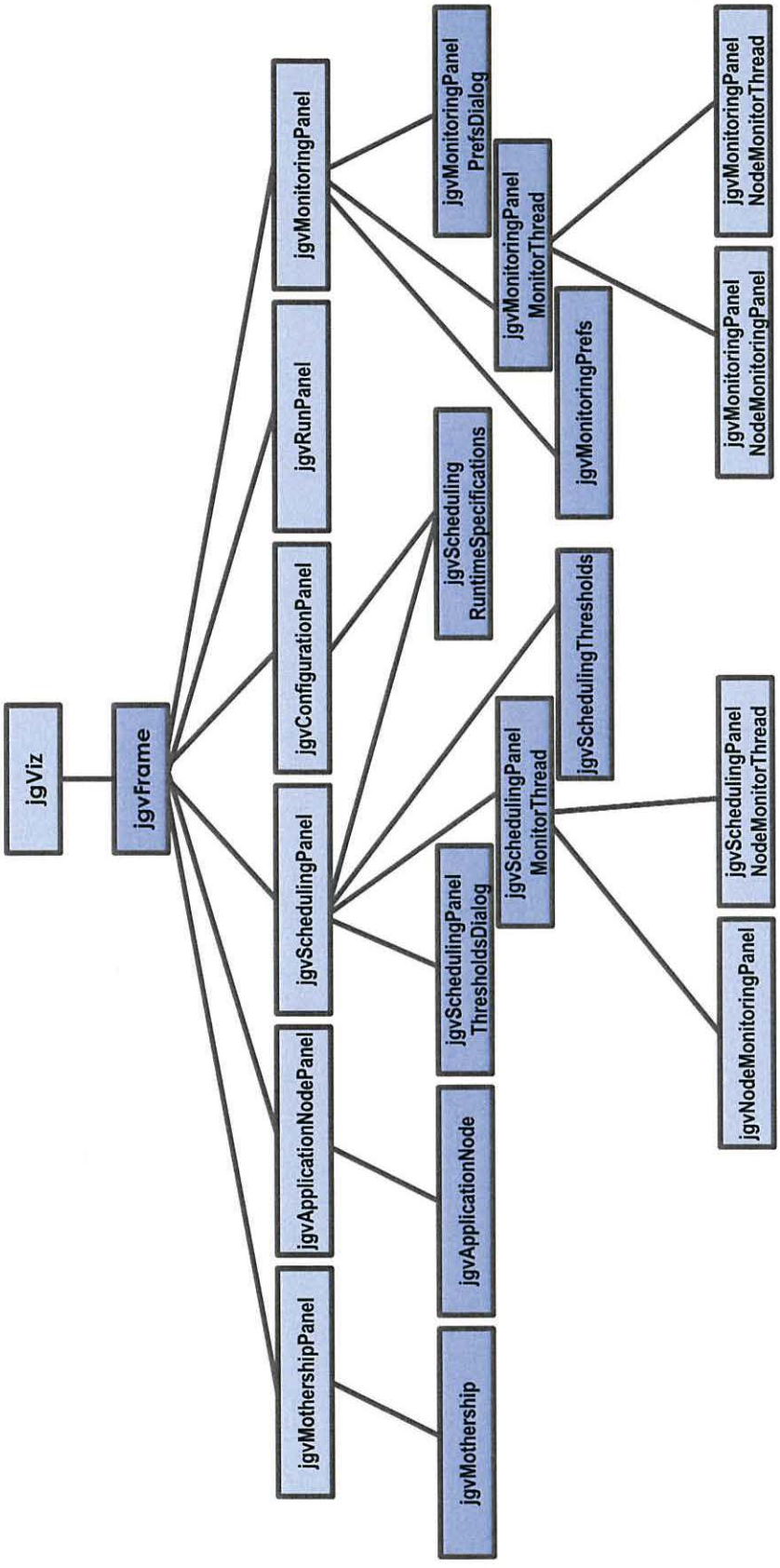


Figure B.1: jgViz Client Class Structure

Appendix C

International Network Topologies

National and International networks, as discussed throughout this thesis, are ever-changing collections of costly and cutting-edge network equipment. Such carrier-grade systems provide the peak of performance for providing connections that form such a major part of the Internet. Of course, networks are only as good as their weakest link and this places additional emphasis on end-to-end solutions, but these networks are the truly enabling factor in geographically dispersed cooperation. We include here, for interest, topology diagrams representing the current state-of-the-art networks that exist. We include commercial ISP networks and then move on to national and international academic networks before finishing with the Global Lambda Integrated Facility - a virtual organization that represents a collection of international networks. For comparison purposes, we also include Jon Postel's original diagram of the arpanet in 1982. How things change.

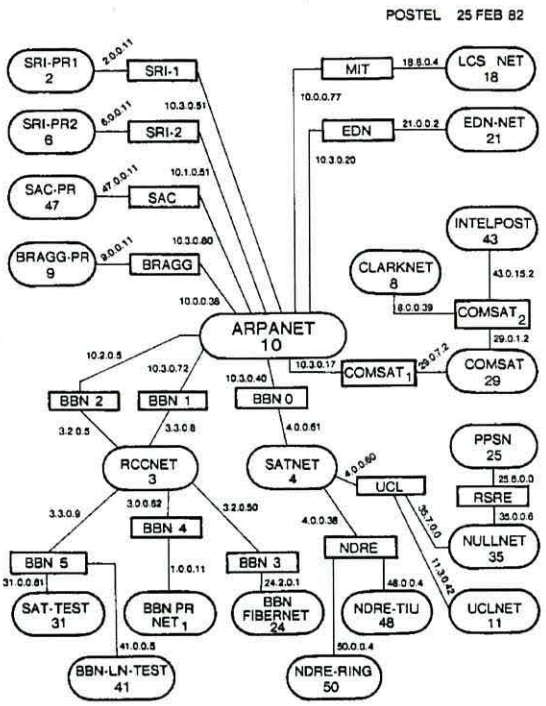


Figure C.1: Arpanet in 1982 by Jon Postel - The Original 'Internet' in One Picture

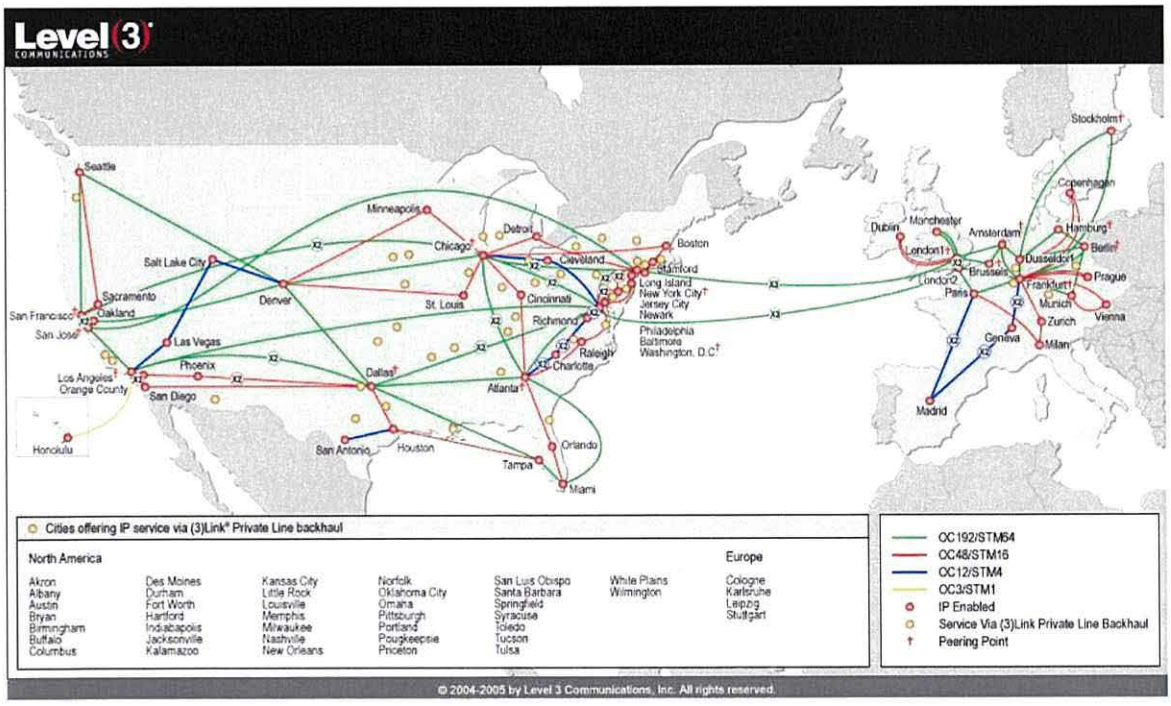


Figure C.2: Level3 ISP International IP Network

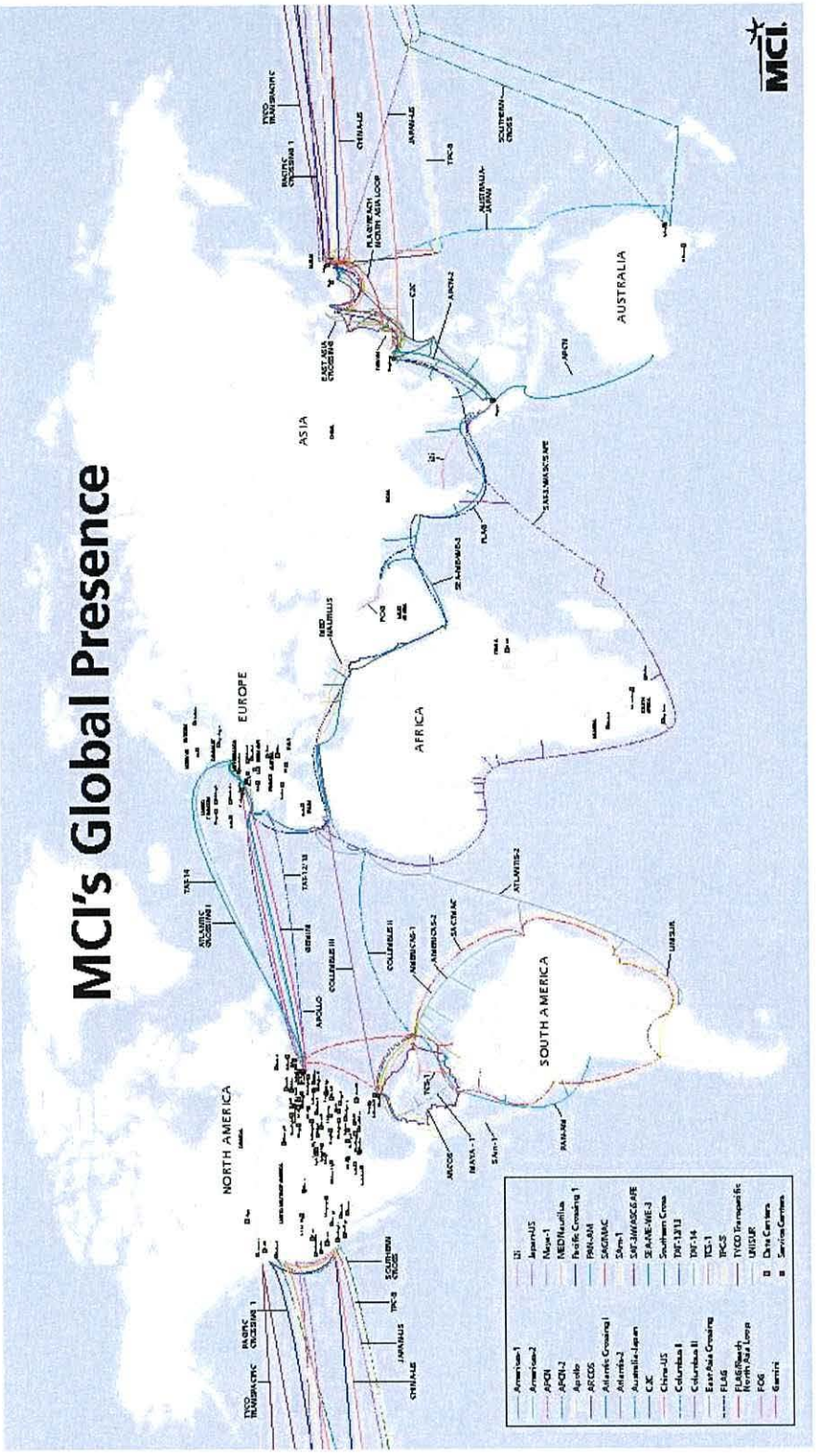


Figure C.3: MCI International IP Network

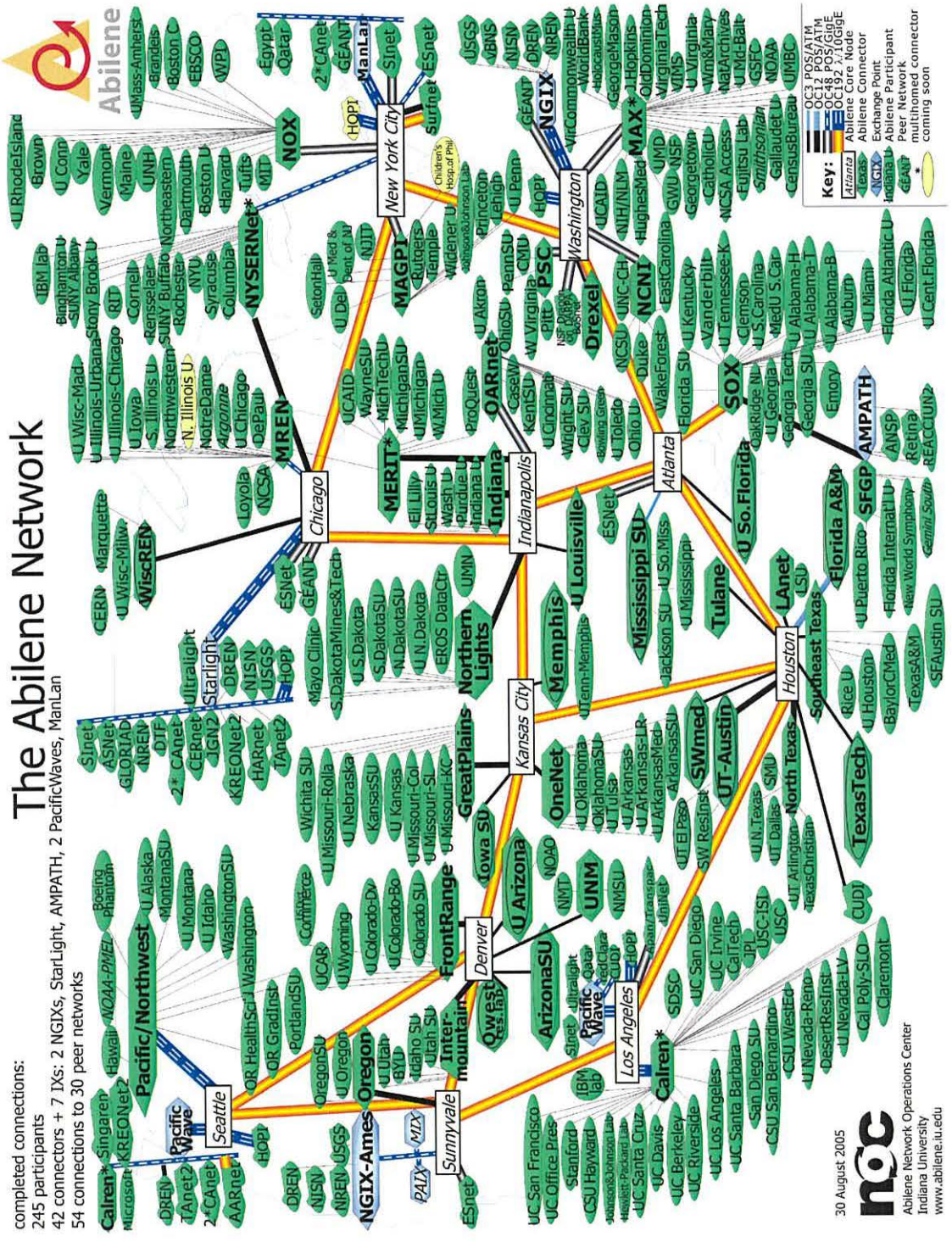


Figure C.4: Abilene 'Internet2' Network in the USA

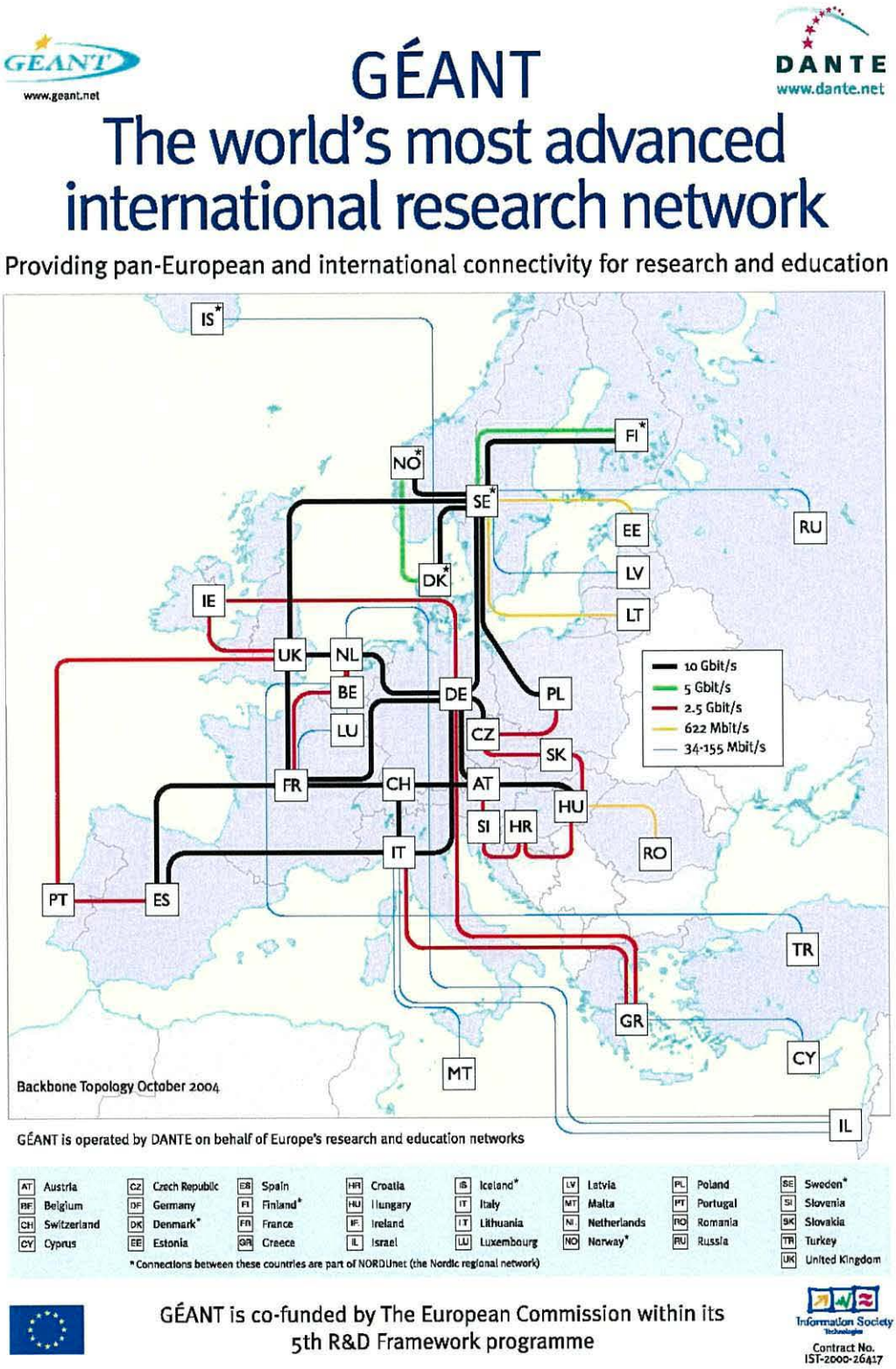


Figure C.5: Geant European Academic Backbone

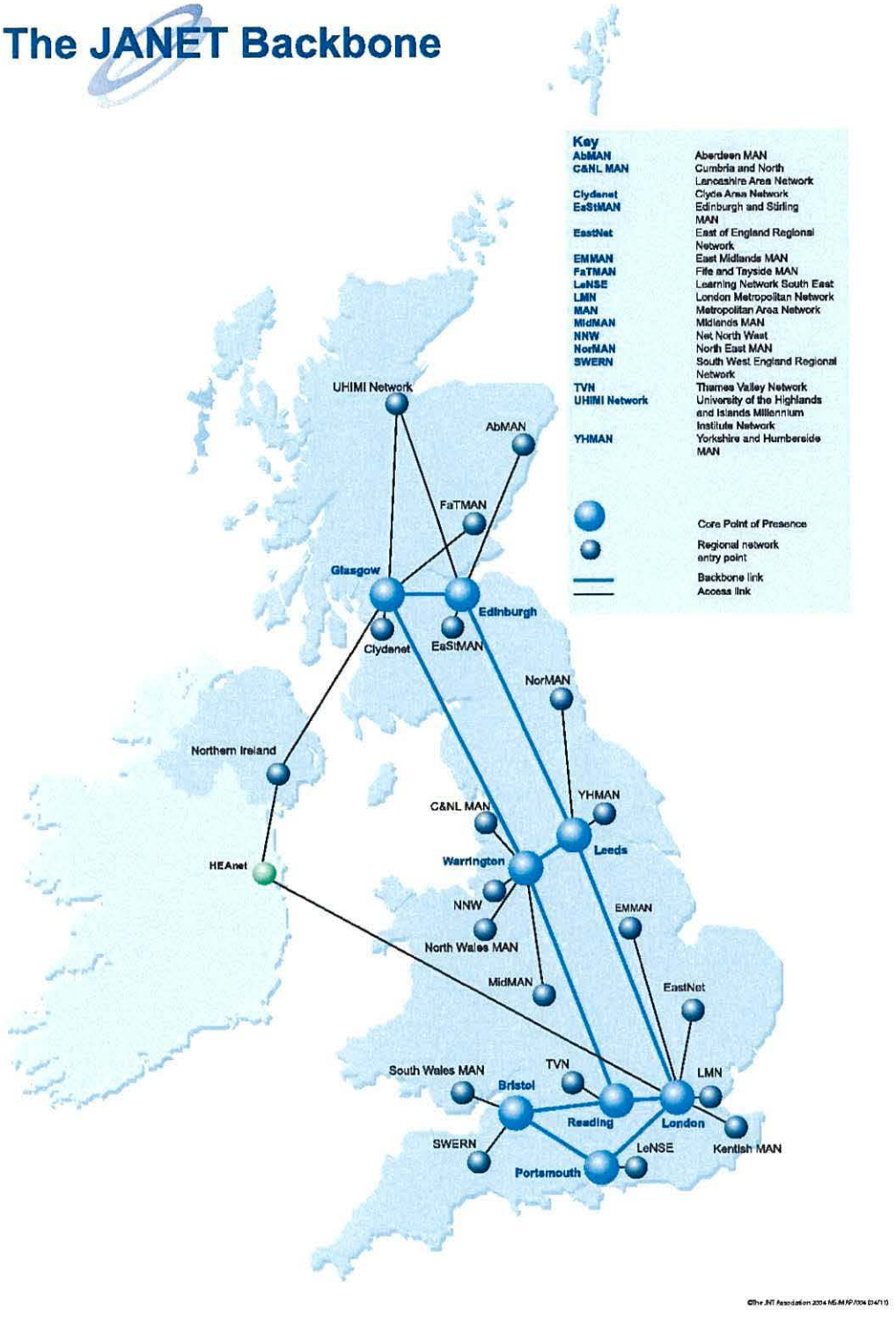


Figure C.6: SuperJanet4 UK Academic Network

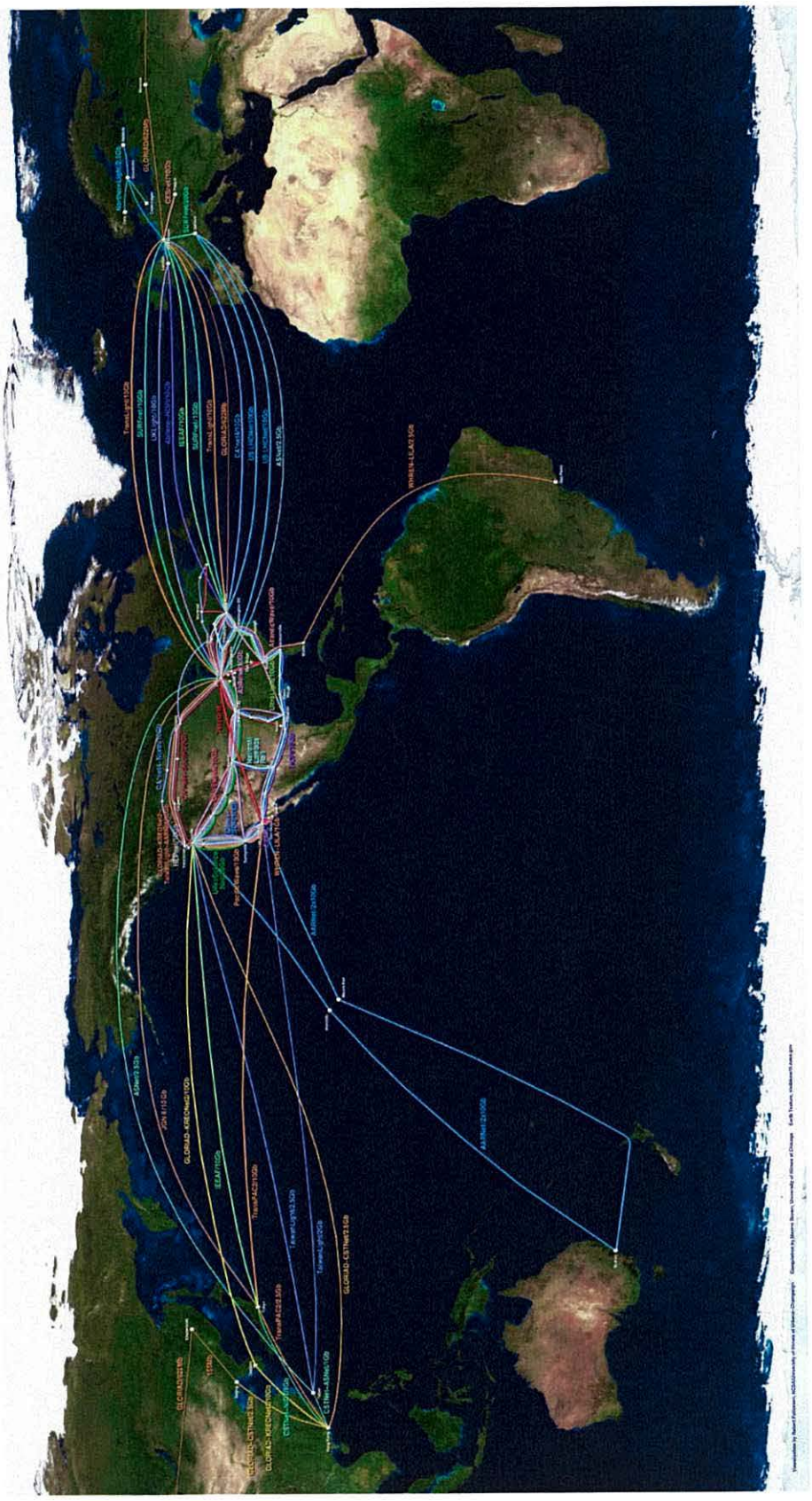


Figure C.7: Global Lambda Integrated Facility Collection of Networks

Appendix D

Poster at Supercomputing

Figure D.1: Poster presented at SuperComputing 2004, 6-12 November 2004, Pittsburgh, USA

References

[1] Fran Berman, Geoffrey C Fox, and Anthony J G Hey, "The grid: Past, present and future," in *Grid Computing: Making the Global Infrastructure a Reality*, Fran Berman, Geoffrey C Fox, and Anthony J G Hey, Eds. Wiley, 2003.

[2] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, "A resource management architecture for metacomputing systems," in *Proceedings IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998, pp. 62–82.

[3] Adrian J Fewings and Nigel W John, "Distributed graphics pipelines on the grid," in *Poster Presentation Proceedings, Supercomputing 2004 Conference*, Pittsburgh, November 2004, University of Wales, Bangor, IEEE/ACM.

[4] M White, G Dunnett, R L Grimsdale, P F Lister, I McGroarty, M D J McNeill, and A D Nimmo, "Workstation graphics-rendering hardware," in *IEE Colloquium on Computer Graphics Systems*. 15 January 1992, pp. 4/1–4/6, IEE.

[5] Brian Paul, "What's new in chromium?," Presentation at Chromium User Group Meeting, Santa Fe, New Mexico, 28 April 2004, Available on the web at `http://www.tungstengraphics.com/SantaFe-BrianPaul_Chromium_User_Mtg/ind%ex.html`.

[6] Jon Louis Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[7] L J Doctor and J G Torborg, "Display techniques for octree-encoded objects," *IEEE Computer Graphics and Applications*, vol. 3, no. 1, pp. 29–38, 1981.

[8] C H Chien and J K Aggarwal, "Volume/surface octrees for the representation of three-dimensional objects," *Comput. Vision Graph. Image Process.*, vol. 36, no. 1, pp. 100–113, 1986.

[9] Jack Veenstra and Narendra Ahuja, "Line drawings of octree-represented objects," *ACM Trans. Graph.*, vol. 7, no. 1, pp. 61–75, 1988.

[10] Aleksander Stompel, Kwan-Liu Ma, Eric B Lum, James Ahrens, and John Patchett, "Slic: Scheduled linear image compositing for parallel volume rendering," in *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. October 2003, IEEE.

[11] K Ma, J S Painter, and M F Krogh, "Parallel volume rendering using binary swap composition," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 59–67, July 1994.

[12] D-L Yang, J-C Yu, and Y-C Chung, "Efficient compositing methods for the sort-last -sparse parallel volume rendering system on distributed memory multicomputers," *The Journal of Supercomputing*, vol. 18, no. 2, pp. 201–220, February 2001.

[13] Gordon Stoll, Matthew Eldridge, Dan Patterson, Art Webb, Steven Berman, Richard Levy, Chris Caywood, Milton Taveira, Stephen Hunt, and Pat Hanrahan, "Lightning-2: a high-performance display subsystem for pc clusters," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. August 2001, ACM.

[14] Shigeru Muraki, Eric B Lum, Kwan-Liu Ma, Masato Ogata, and Xuezhen Liu, "A pc cluster system for simultaneous interactive volumetric modeling and visualization," in *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. October 2003, IEEE.

[15] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan, "Wiregl: A scalable graphics system for clusters," in *Proceedings of SIGGRAPH 2001*. 2001, ACM.

[16] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D Kirchner, and James T Klosowski, "Chromium: A stream-processing framework for interactive rendering on clusters," in *Proceedings of SIGGRAPH 2002*. 2002, ACM.

[17] Brian Paul, "Scalable rendering with chromium," Web Presentation, http://www.tungstengraphics.com/chromium/chromium.html.

[18] Oliver G. Staadt, Justin Walker, Christof Nuber, and Bernd Hamann, "A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering," in *EGVE '03: Proceedings of the workshop on Virtual environments 2003*. 2003, pp. 261–270, ACM Press.

[19] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh, "Hybrid sort-first and sort-last parallel rendering with a cluster of pcs," in *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. 2000, pp. 97–108, ACM Press.

[20] Jian Yang, Jiaoying Shi, Zhefan Jin, and Hui Zhang, "Design and implementation of a large-scale hybrid distributed graphics system," in *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*. 2002, pp. 39–49, Eurographics Association.

[21] C. Winkelholz and T. Alexander, "Approach for software development of parallel real-time ve systems on heterogenous clusters," in *EGPGV 02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*. 2002, pp. 23–32, Eurographics Association.

[22] P.G. Lever, G.W. Leaver, I. Curington, J.S. Perrin, A.W. Dodd, N.W. John, and W.T. Hewitt, "Design issues in the avs/express multi-pipe edition," in *IEEE Visualization 2000*, October 2000.

[23] ModViz Inc, "Virtual graphics platform (vgp) data sheet," Web site, January 2005, http://www.modviz.com.

[24] "Sun Microsystems Inc.", "Sun fire visual grid," Web site, January 2004, http://www.sun.com/solutions/hpc/compute/visualgrid/.

[25] IBM, "Deep computing visualization: Gain insight into your data," Web site, January 2005, http://www-1.ibm.com/servers/deepcomputing/visualization/.

[26] Hewlett Packard, "Scalable high-performance visualization system," Web presentation, January 2005, http://www.hp.com/techservers/hpccn/sci_vis/sepia_vis_presentation.pdf.

[27] Thomas A. DeFanti, Ian Foster, Michael E. Papka, Rick Stevens, and Tim Kuhfuss, "Overview of the i-way: Wide-area visual supercomputing," *International Journal of Supercomputer Applications and High Performance Computing*, vol. 10(2), pp. 123–130, 1996.

[28] J Jamison and R Wilder, "vbns: the internet fast lane for research and education," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 60–63, January 1997.

[29] Ian Foster, "The grid: A new infrastructure for 21st century science," *Physics Today*, vol. 55, no. 2, pp. 42, February 2002.

[30] David De Roure, Mark A Baker, Nicholas R Jennings, and Nigel R Shadbolt, "The evolution of the grid," in *Grid Computing: Making the Global Infrastructure a Reality*, Fran Berman, Geoffrey C Fox, and Anthony J G Hey, Eds., Wiley Series in Communications Networking and Distributed Systems, chapter 3, pp. 65–100. Wiley, 2003.

[31] Ian Foster, Jonathan Geisler, Bill Nickless, Warren Smith, and Steven Tuecke, "Software infrastructure for the i-way high-performance distributed computing environment," in *Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*, August 1996, pp. 562–571.

[32] FAFNER RSA Factoring Project, "http://www.npac.syr.edu/factoring.html," Web site.

[33] E Korpela, D Werthimer, D Anderson, J Cobb, and M Leboisky, "Seti@home-massively distributed computing for seti," *Computing in Science & Engineering*, vol. 3, no. 1, pp. 78–83, Jan-Feb 2001.

[34] Ian Foster and Carl Kesselmann, Eds., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kauffmann, 1998.

[35] M Romberg, "The unicore architecture: seamless access to distributed resources," in *The Eighth Internation Symposium on High Performance Distributed Computing*, 1999.

[36] A S Grimshaw, W A Wulf, and the Legion Team, "The legion vision of a worldwide virtual computer," *Communications of the ACM*, vol. 40, no. 1, pp. 39–45, January 1997.

[37] M J Litzkow, M Livny, and M W Mutka, "Condor-a hunter of idle workstations," in *Distributed Computing Systems, 1988., 8th International Conference on*, June 1988, pp. 104–111.

[38] World Wide Web Consortium, "Web services description language draft note," Web draft, February 2006, `http://www.w3.org/TR/wsdl`.

[39] K Brodlie, J Brooke, M Chen, D Chisnall, A Fewings, C Hughes, N W John, M W Jones, M Riding, and N Roard, "Visual supercomputing - technologies, applications and challenges," in *25th Annual Conference of the European Association for Computer Graphics*, Christophe Schlick and Werner Purgathofer, Eds. The Eurographics Association, August-September 2004.

[40] Organization for the Advancement of Structured Information Standards (OASIS), "Web services reference framework specification," Web site, February 2006, `http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsrf`.

[41] Ian Foster, "Globus toolkit version 4: Software for service-oriented systems," in *IFIP International Conference on Network and Parallel Computing*, 2005, pp. 2–13.

[42] Ian Foster and Carl Kesselman, "Globus: A metacomputing infrastructure toolkit," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, pp. 115–128, Summer 1997.

[43] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, and S. Tuecke, "Data management and transfer in high performance computational grid environments," *Parallel Computing Journal*, vol. 28, no. 5, pp. 749–771, May 2002.

[44] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke, "Gass: A data movement and access service for wide area computing systems," in *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, 1999, pp. 78–88.

[45] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman, "Grid information services for distributed resource sharing," in *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, August 2001.

[46] Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane, "A Java Commodity Grid Kit," *Concurrency and Computation: Practice and Experience*, vol. 13, no. 8-9, pp. 643–662, 2001.

[47] Gregor von Laszewski, Jarek Gawor, Sriram Krishnan, and Keith Jackson, "Commodity grid toolkits - middleware for building grid computing environments," in *Grid Computing: Making the Global Infrastructure a Reality*, Fran Berman, Geoffrey C Fox, and Anthony J G Hey, Eds. Wiley, 2003.

[48] Nasa, "http://www.nas.nasa.gov/about/projects/columbia/columbia.html," Web site, September 2005.

[49] The BlueGene/L Team, "An overview of the bluegene/l supercomputer," Tech. Rep. 0-7695-1523-X, IBM Research and IBM Rochester and the Lawrence Livermore National Laboratory, 2002.

[50] GLIF, "http://www.glif.is/," Web site, September 2005.

[51] William E Johnston, *Grid Computing Implementing production Grids*, chapter 5, pp. 117–167, Wiley, 2003.

[52] John D Ainsworth and John M Brooke, "Testing for scalability in a grid resource usage service," in *Proceedings of the UK e-Science All Hands Meeting 2005*. EPSRC, 19-22 September 2005.

[53] Pratap Pattnaik, Kattamuri Ekanadham, and Joefon Jann, *Grid Computing Autonomic Computing and the Grid*, chapter 13, pp. 351–361, Wiley, 2003.

[54] Craig Lee and Domenico Talia, *Grid Computing Grid programming models: current tools, issues and directions*, chapter 21, pp. 555–578, Wiley, 2003.

[55] Charles Hansen, "Known and potential high performance computing applications in computer graphics and visualization," in *High Performance Computing for Computer Graphics and Applications*, M. Chen, P. Townsend, and J.A. Vince, Eds. July 1995, Proceedings of the International Workshop on High Performance Computing for Computer Graphics and Visualization 1995, pp. 23–29, Springer.

[56] Thierry Benoist, W T Hewitt, and Nigel W John, "Corba visualization platform," in *Short Presentations Proceedings, Eurographics 2001*, September 2001, ISSN 1017-4656.

[57] Olivier Martin, "The eu datatag project," Tech. Rep. GGF3, Global Grid Forum, Frascati, Italy, 2001.

[58] L Renambot, T van der Schaaf, H Bal, D Germans, and H J W Spoelder, "Griz: Experience with remote visualization over an optical grid," *Journal of Future Generation Computer Systems (FGCS)*, vol. 19, no. 6, pp. 871–882, August 2002.

[59] Gabrielle Allen, Werner Benger, Tom Goodale, Hans-Christian Hege, Gerd Lanfermann, Andr? Merzky, Thomas Radke, Edward Seidel, and John Shalf, "The cactus code: A problem solving environment for the grid," in *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC9)*, 2000.

[60] Gabrielle Allen, Werner Benger, Tom Goodale, Hans-Christian Hege, Gerd Lanfermann, Andr? Merzky, Thomas Radke, Edward Seidel, and John Shalf, "Cactus tools for grid applications," *Cluster Computing*, vol. 4, pp. 179–188, 2001.

[61] J. M. Brooke, P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning, and A. R. Porter, "Computational steering in realitygrid," in *Proceedings of the UK e-Science All Hands Meeting 2003*, 2003.

[62] S.M. Pickles, R.J. Blake, B.M. Boghosian, J.M. Brooke, J. Chin, P.E.L. Coveney, R. Haines, J. Harting, M. Harvey, S. Jha, M.A.S. Jones, M. McKeown, R.L. Pinning, A.R. Porter, K. Roy, and M. Riding, "The teragyroid experiment," in *GGF10: Workshop on Case Studies on Grid Applications*, March 2004.

[63] Paul Heinzlreiter and Dieter Kranzlmuller, "Visualization services on the grid: The grid visualization kernel," *Parallel Processing Letters*, vol. 13, no. 2, pp. 135–148, 2003.

[64] D. Kranzlmuller, G. Kurka, P. Heinzlreiter, and J. Volkert, "Optimizations in the grid visualization kernel," in *PDIVM 2002, Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia*, 2002.

[65] Ian J Grimstead, Nick J Avis, David W Walker, and Roger N Philip, "Resource-aware visualization using web services," in *Proceedings of the UK e-Science All Hands Meeting 2005*. EPSRC, 19-22 September 2005.

[66] Ian J. Grimstead, Nick J. Avis, and David W. Walker, "Automatic distribution of rendering workloads in a grid enabled collaborative visualization environment," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2004, IEEE Computer Society.

[67] David Hughes, "Sinking in the sea of pixels - the case for media fusion," White paper, SGI, Corporate Office 1500 Crittenden Lane Mountain View, CA 94043, 2005.

[68] John Domingue, Blaine A. Price, and Marc Eisenstadt, "A framework for describing and implementing software visualization systems," in *Graphics Interface '92*, 1992, pp. 53–60.

[69] gViz Team, "The gviz web site http://www.comp.leeds.ac.uk/vis/gviz," Web Site, March 2005.

[70] D A Duce and M Sagar, "skml a markup language for distributed collaborative visualization," in *Eurographics UK Theory and Practive of Computer Graphics 2005*, L Lever and M McDerby, Eds., 2005.

[71] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs, "A sorting classification of parallel rendering," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 23–32, July 1994, University of North Carolina at Chapel Hill and Princeton University.

[72] G. VoB;, J. Behr, D. Reiners, and M. Roth, "A multi-thread safe foundation for scene graphs and its extension to clusters," in *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*. 2002, pp. 33–37, Eurographics Association.

[73] E. Wes Bethel, Greg Humphreys, Brian Paul, and J. Dean Brederson, "Sort-first, distributed memory parallel visualization and rendering," in *Proceedings of the 2003 IEEE Symposium on Parallel and Large Data Visualization and Graphics [PVG 2003]*. 2003, IEEE.

[74] R Scheifler, "The x windows system," *ACM Transactions on Graphics*, 1986.

[75] The X Consortium, "The xhost man page: http://www.xfree86.org/current/xhost.1.html," Web Site, September 2005.

[76] The X Consortium, "The xauth man page: http://www.xfree86.org/4.4.0/xauth.1.html," Web site, September 2005.

[77] Tungsten Graphics Inc., "The dri website: http://dri.freedesktop.org/wiki/," Web site, September 2005.

[78] nVidia Corporation, "http://www.nvidia.com/content/drivers/drivers.asp," Web site, September 2005.

[79] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanna Yoakum-Stover, "Gpu cluster for high performance computing," in *Proceedings of the ACM/IEEE Supercomputing Conference, 2004.*, November 2004, pp. 47–51.

[80] Jin Young Hong and May D Wang, "High speed processing of biomedical images using programmable gpu," in *International Conference on Image Processing, ICIP '04.*, October 2004, vol. 4, pp. 2455–2458.

[81] George Kola, Tevfik Kosar, and Miron Livny, "Profiling grid data transfer protocols and servers," in *Proceedings of 10th European Conference on Parallel Processing (Euro-Par 2004)*, 2004.

[82] Rajkumar Kettimuthu, "Globus striped gridftp framework and server," Web Presentation, August 2005, Department of Computer Science and Engineering, The Ohio State University, Columbus, Ohio.

[83] Anjan Pakhira, Ronald Fowler, Lakshmi Sastry, and Toby Perring, "Grid enabling legacy applications for scalability - experiences of a production application on the uk

ngs," in *Proceedings of the UK e-Science All Hands Meeting 2005*. EPSRC, 19-22 September 2005.

[84] Rick 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*, vol. 15, no. 5-6, pp. 119–132, October 1999.

[85] Mark Leese, Rik Tyler, and Robin Tasker, "Network performance monitoring for the grid," in *Proceedings of the UK e-Science All Hands Meeting 2005*. EPSRC, 19-22 September 2005.

[86] Plusplus, "Roller coaster 2000," Web site, August 2005, http://plusplus.free.fr/rollercoaster/.

[87] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal, "Dynamic instrumentation of production systems," in *USENIX 2004 Annual Technical Conference*, 2004, pp. 15–28.

[88] Sachin Wasnik, Mark Prentice, Noel Kelly, P V Jithesh, Paul Donachy, Terence Harmer, Ron Perrott, Mark McCurley, Michael Townsley, Jim Johnston, and Shane McKee, "Resource monitoring and service discovery in genegrid," in *Proceedings of the UK e-Science All Hands Meeting 2005*. EPSRC, 19-22 September 2005.

[89] M Riding, J D Wood, K W Brodlie, J M Brooke, M Chen, D Chisnall, C Hughes, N W John, M W Jones, and N Roard, "e-viz: Towards an integrated framework for high performance visualization," in *Proceedings of the UK e-Science All Hands Meeting 2005*. EPSRC, 19-22 September 2005.

The end of the roller coaster ride