The Monte Carlo Array Processor as a Grid Resource

David Edward Laurence Jones

September 2005

University of Liverpool

Department of Physics

Thesis submitted to University of Liverpool for the degree of Doctor of Philosophy

Dedication

Jamie Peter Wigley

14th February 2003 - 12th September 2003

Acknowledgements

I would like to acknowledge the help and support of many people during the research and writing of this thesis. Undoubtedly I will forget somebody so I will apologise in advance.

Department of Physics, Liverpool University:

Tony Moreton, Themis Bowcock, Mike Houlden, Tracey Berry, Barry King, the late Paul Booth, Michael George, Andy Washbrook, Mark Tobin, Richard Sloane, Girish Patel, Peter Rowlands, Brian Rae, Anant Gajjar, Adrian Bevan and Katherine George, Carlos Chavez, Dave Payne, Martin Turner, Richard Parry, Helen Hayward, Arwel Evans, Steve Maxfield.

Family and Friends:

Mum and dad (Hilary and Richard), brother (Christopher), sister (Heather), Nicola, Natasha, Ben and Lisa. Robert and Maria-Olga (my extra parents in Geneva). Mercedes, Franscisco (including Fransoir), Eric, Monica and family. Grandma, Alan-John, Alan and Peggy Mister, Grandad Jones.

Neston County High School. 'The Slac'. Chris Duncan, Simon Finnigan, Toby Mercer, Neesha Arulnagagen.

CERN, Geneva:

Susan Cannon, Mary Elizabeth Shewry, Omar Sharif, Leo Jenner, Peter Kunszt, Leanne Guy, Eric Grancher, George Shering.

OAC:

Christopher Weatherby (and his numerous adventures!) and everyone else.

Contents

List of Figures

Abstract

This thesis describes the implementation of a proof of concept *web application* (web enabled software program) using *Spitfire.* Spitfire is a software product of the EU-Datagrid. The system was designed to provide a web-based browser and administration tool for the *Spitfire system.*

The *Monte-Carlo Array Processor* (MAP) at the University of Liverpool is a *commodity* supercomputer. The large number of nodes provides massive parallel processing capabilities. Described in this thesis is a *web application* called DUCK. This was designed to turn MAP into a *resource* that can be accessed from the *world wide web.* The system uses *servlet* technologies provided by the Java platform. Persistency is provided by *relational databases.* First steps were made to turn MAP into a *grid resource.*

Chapter 1

Introduction

The *Large Hadron Collider* (LHC) is a particle accelerator located at CERN[1] which is on the Franco-Swiss border near Geneva. It is due to become operational in 2007, when it will begin colliding beams of protons together at a centre of mass energy of 14TeV. Each of the proton-proton collisions will, in general, create a large number of secondaries that will need reconstructing. Reconstruction is the process of taking the raw data and deducing what actually happened in the collision. Beams collide at each of the 4 interaction points at 40 MHz. Detectors will record these collisions, with 15 million Gigabytes of data being produced per year [2]. Scientists working on LHC are from institutions in many countries and they will need access to these data sets concurrently from geographically distributed locations. The computing technologies that have been used for this task in the past are not sufficient. Hence new forms of computing technology, such as grids which are discussed in chapter 2, are needed.

This thesis is divided into two parts. In the first part I discuss and explore why grids are required and the theory behind them. A more formal description of data, structure and storage is given. The hardware and software that make modern computing possible are also briefly described. In the final chapter of this first part I discuss '*cluster and network computing*', including the underlying technologies and the higher level abstractions. The concepts and

technologies of *web services* are covered.

The original research is described in the second part of the thesis, starting with the chapter entitled 'Spitfire'. This covers my work for the EU-Datagrid project. The product of this work was a web-based administration tool for relational databases. The largest contribution to my research is my development of 'DUCK' and a description of how it can be made into a grid resource. DUCK is a web-interface for the *Monte-Carlo Array Processor*(MAP) at the University of Liverpool.

Chapter 2

Grid Principles

2.1 Introduction

The term grid or computing grids[3],[4] have been used to describe many concepts and has been attached to the marketing of many products. Behind the expectations, there is a set of technologies that can claim to be grid-like. These are already being used in applications for biology, health, physics and many other projects. Broadly there are three major concepts behind a grid.

Firstly, grids aim to change the way in which we share resources. The internet[5], specifically the world wide web[6], has revolutionised the way we share and distribute information. With the world wide web we have a collection of distributed documents (typically web pages). Whereas with the grid we aim to have a distributed collection of computing resources.

Secondly, grids aim to eliminate the reliance on a single central resource and they are largely viewable as decentralised. People and institutions need to share computing resources because of the demands of modern science and industry. Many problems require vast amounts of computing cycles, storage space or specialised equipment. It can be difficult to provide these in one location. Existing high performance computing (HPC) often works in a manner that is highly centralised.

Thirdly, a unique aspect of the decentralised nature of grids is the inhomogeneity of the systems that they are composed of. It is almost impossible to impose homogeneity over continents, countries and different institutions.

However, it should not be supposed that grid are completely new. Most grid technologies are built upon existing standards and solutions. In addition, grids must map onto existing resources and ways of working. It is useful to consider how grids may be used in some applications.

2.2 Applications

The success of any computing innovation is dependent on it not being a stand alone innovation without any applicability for use in the real world. Grids were created with the requirements and demands of scientists in mind. Demands for distributed computing vary widely between the different user groups[7], the following examples illustrate this diversity.

One example is *simulation* or models of physical systems. These can be very compute intensive as often the same simulation is run millions of times with different input parameters. Alternatively a simulation may be repeated to gain understanding of stochastic processes. Such simulations tend to be run in parallel on batch systems[1]. In general the batch jobs do not communicate with each other. Monte Carlo events in high energy physics are often generated in this way.

Another use case is *interactive applications.* Here the ability to run extremely powerful applications is linked with the visualisation and control of these applications in real time. Examples include image processing for satellites and microscope or medical applications, such as computer aided surgery. Such examples require a high quality of service.

Other computing applications require large amounts of processing power for

[1]A *batch system* is a computer system (usually multiple machines) intended to provide a pool of computing power. Users typically submit jobs to a batch system. The system is responsible for scheduling, staging and executing the jobs. For an example see [8]

short periods of time to meet a specific deadline; an examples of which is the financial industry. In cases like this scheduling of compute cycles is critical.

Finally, there are problems, in astronomy and physics, where the analysis is done over huge, distributed sets of data. These problems include bandwidth over network connection, authorisation at multiple sites and multiple users accessing data.

2.3 Virtual Organisations

More recently the concept of virtual organisations (VOs) has become central to grid computing[9]. VOs consist of groups of people/institutions sharing resources and using a set of defined rules for sharing. Different VOs will, in general, tend to run different applications and have their own set of requirements.

VOs may be geographically distributed, use different computing hardware, be members of different companies or institutes and have expertise in different areas. They can also be rapidly created or reconfigured. Examples of VOs in HEP are ALICE[10], ATLAS[11], CMS[12] and LHCb collaborations [13].

The dynamic sharing of resources differentiates a grid from the web or a traditional collaboration. Resources can be computing power, programs, sensors and services. All of these resources are available, subject to rules, to members of the VO.

2.4 Infrastructure

For economical reasons grids tend to re-use the 'Internet' infrastructure or fabric. For example there are not necessarily separate 'grid wires' or 'grid backbones'. Existing high speed networks are used. In addition existing servers make up part of the grid at local level. Grid has to co-exist with these servers and, ideally, need minimal changes to their existing configuration.

In particular existing end user systems should not need to be replaced. This provides maximum accessibility to grid technology. Thus grids will have to work with heterogeneous *commodity* clients (see Appendix F.3), Linux, Windows and Mac OS 'boxes'. In terms of software infrastructure; there needs to be a central core set of software components called *middeware*; which provides the functionality for the grid and provides a common basis for the *application programming interface*(API) to the developer. Middleware can be thought of as the glue for putting grids together.

Middleware implementations include software in the *Globus Toolkit* [14], *Condor* [15] and *Legion*[16]. These toolkits enable grid systems to be built and applications to be grid enabled.

2.5 Layered Model

The ideas that have been discussed above have be formalised in what is known as a 'layer model'. The paper 'Anatomy of the Grid'[17] splits grid architecture into five layers, as illustrated in figure 2.1. The layers are built up on top of the fabric layer finishing with the application layer at the top.

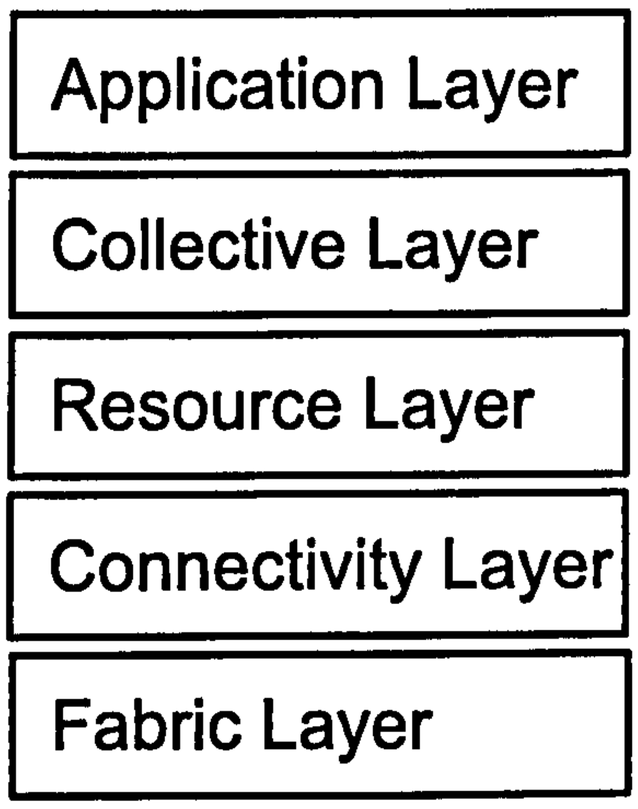

Figure 2.1: Diagram depicting the 5 main layers in grid architecture.

A more detailed view of the layer architecture is shown in the figure 2.2. Typically users will only concern themselves with the top layer, the application layer. Below we discuss the function of the 5 layers.

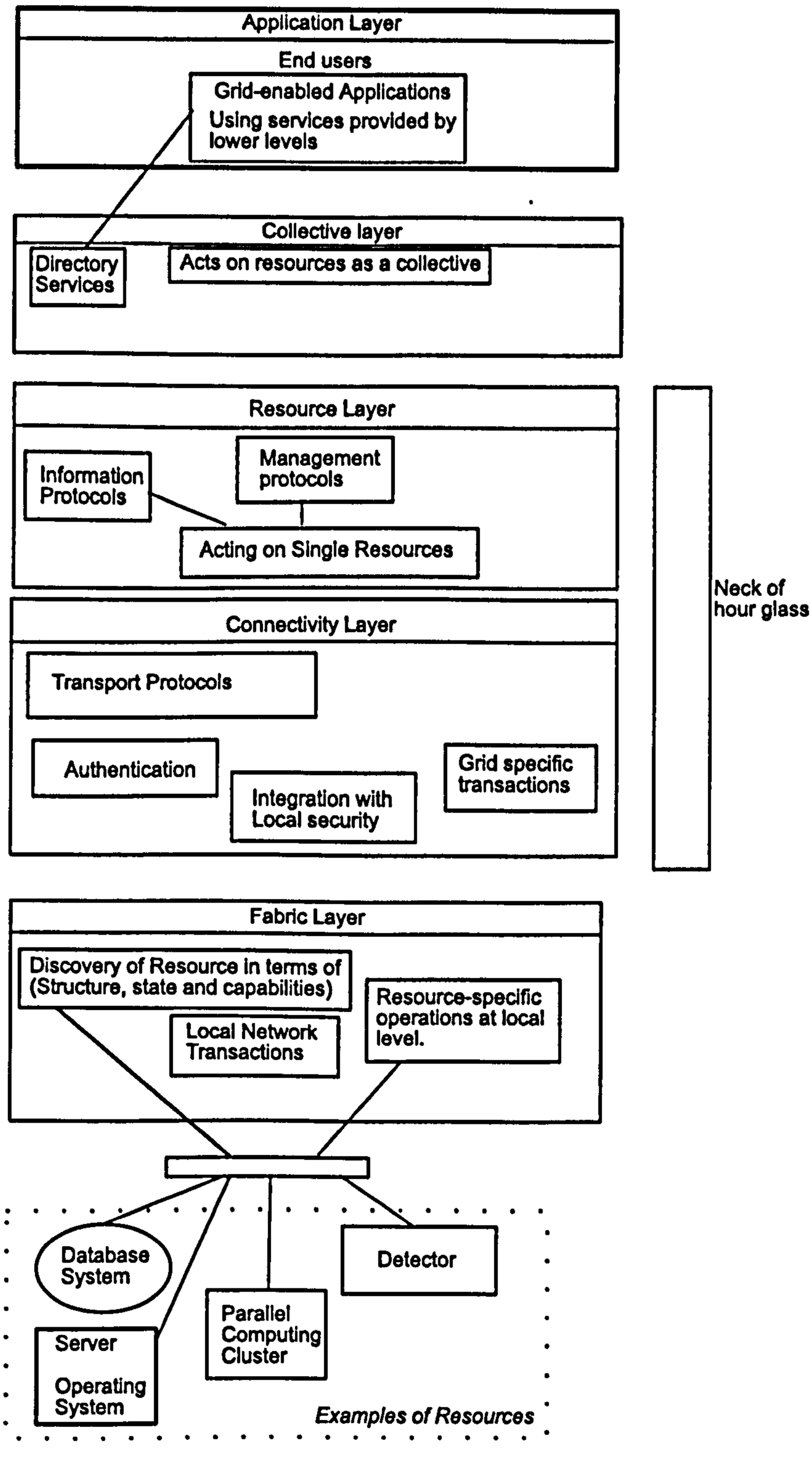

Figure 2.2: Diagram depicting the 5 main layers in grid architecture, showing some selected components..

2.5.1 Fabric Layer

The *fabric layer* can be thought of as presenting the local resources to the grid, such as CPU farms or storage. The components of this layer are hardware and reside locally at the site of the resource. Resources are to be presented to the grid via the software this layer provides. If this software provides a rich set of operations, deployment becomes increasingly difficult. Conversely, the less functionality the software provides, the simpler the deployment.

The implementation of the components that build up the resource layer can involve internal protocols which are not visible to the rest of the grid architecture. Examples of such protocols are NFS access protocols, database connectivity (ODBC and JDBC are examples, see 4.7.4) and batch processing control protocols, LSF[18], PBS[19].

Enquiry mechanisms allow the grid to discover the services that are available to it. It has been suggested[20] that a fabric enquiry mechanism should at least provide the following information for a resource. This software should allow the discovery of the *resource structure* and *characteristics*, additionally, it should provide the *current state* of the resource. An example of this, for a storage resource, would be the amount of space available and the amount of bandwidth available to the resource. Finally, a method to provide discovery of the *capabilities* of the resource should be provided. An example of this, for a computing resource, would be the total amount of processing power available.

The following subsection gives examples of elements that are considered resources on the grid. They are typically elements that would reside in the *fabric layer.*

Storage Resources

Typically, storage resources fall into 3 categories, disk pools, tape repositories and optical disk repositories.

Pools of hard disks offer fast access to large amounts of data. There are various

methods of ensuring data is not lost due to hardware failure; examples being *Redundant Array of Inexpensive Disks*(RAID), see [21]) and '*geoplexing*', which is making a backup to a geographically separate location.

Tape repositories can be used for large datasets. *Tape robots* are used to retrieve the required tapes from storage and mount the tape to be read. Access times to data are slower due to, firstly, that the tape robot will need to retrieve the tape on which the data is on and secondly tape read times tend to be slower. *Block positioning* means the majority of the data on the tape need not be read to retrieve a given portion.

Optical disk repositories consisting of *optical media* can be used to store large datasets in a similar manner to tape systems.

For all systems, mechanisms are required that allow the reading and writing of files. Replication of frequently used/heavily used data sets may also be required.

Computational Resources

Examples of computational resources on the grid are computer clusters, groups of machines and the computational resources behind batch systems.

Computer clusters consist of arrays of computers connected in private local networks that allow problems to be solved using parallel processing techniques. Groups of individual computers, for example desktop computers in offices, can provide computational cycles when they are not being used (typically at night).

The computational resources behind batch systems are important resources. Batch systems provide management of computing resources that allow batch jobs to be run. Batch jobs are distinct from interactive jobs, where the user directly controls the computation.

In all cases mechanisms are required to allow jobs to be prioritised, setup, started, monitored, finished and the resulting output obtained.

Other Resources

Some additional examples of grid resources are: -

- *Networks.* Network transfers need managing depending on criteria such as network load and capacity.
- *Code Repositories.* Repositories can hold programming source code for large projects. Typically versioning and auditing are supported.
- *Catalogs.* Specialised forms of storage for structured data to support update and query operations. Typically these resources are implemented using a relational database.

2.5.2 Connectivity Layer

Communication and authentication protocols are defined by the connectivity layer. This provides data exchange between grid resources, include *transport*, *routing* and *naming*. Typical grid implementations use protocols from the TCP/IP stack.

The requirements of the authentication part of the connectivity protocol are heavily security related and include single sign on, delegation, using existing security installations and trust relationships.

Single sign on means that once users are signed on to the grid, they should not need to authenticate again to access different resources on the grid.

Delegation means the user can allow a program to run with the user's authentication. This means the program can access the resources the user is authorised to use. Delegation allows a program to further delegate these rights to another program.

The ability to use existing security systems at local sites is required. The connectivity layer must inter-operate with existing security solutions and not require new systems to be installed at the local site, therefore it must inter-operate with a large range of potential security protocols.

Trust relationships build on user rights. If a user has the right to use resources at two sites separately then the user should be able to use the resources of both sites together without the sites needing to interact.

Most of the above issues are related to security; these security issues are described in greater detail below. See the papers "A Security Architecture for Computational Grids" [22] and "The Security Architecture for Open Grid Services" [23] for more detailed discussions of potential grid security architectures.

Security

Sites on the world wide web typically have public pages, which are visible by the public with little restriction. Some sites use user/password authorisation to restrict content to privileged users. For example, a user may have to register to access content that might be confidential or of monetary value (pay per view).

With few exceptions security authorisations are restricted to one site; meaning users need to login to each site they wish to use.

In contrast, the challenges faced to provide grid security are a lot more complex. Typically the challenges fall into 3 areas, *integration*, *interoperability* and *trust relationships.*

Integration

Integration means that a grid security solution must work with existing, non-grid, security implementations. Resources and domains that are part of a grid will have their own existing security solutions. It is not feasible or desirable to replace these solutions, so grid implementations must be integrated to work with them.

Certain *firewall issues* arise when sites and individual machines are fire-walled from the network for security reasons. It needs to be ensured that services are not blocked by the actions of firewalls behaving in this way.

Interoperability

Interoperability is an important aspect of security on the grid because using the grid may involve using many domains and different environments, security schemes need to interoperate. There must be ways for the environments to exchange security information. Interoperability issues split up into ones of protocol, policy and interoperability across domains.

At the protocol level, the actual process of exchanging information, we need to consider the methods used to exchange the data. This means the transport medium; examples of such are *HTTP* or *SOAP* (see I.6). *Confidentiality* is an issue here; users need to be able to rely on the underlying protocols to protect the confidentially of the exchanged messages and documents. There are various means of encrypting HTTP and SOAP traffic to achieve this. *Integrity* is assured by detecting unauthorised changes to messages whilst they are being transported. Such changes are often introduced in a so-called 'man in the middle attack' performed by a third party. Protection against such incidents can be provided by using public-private keys and other similar mechanisms[24].

At policy level there must be ways to exchange and understand other entities security policies. This is because security arrangements may involve multiple parties and domains. *Identities* must be transferable between systems; meaning there must be the ability to identify users across different domains. This involves the mapping of user identity and user credentials across domain boundaries. This falls under the issue of *Authorization*, which is whether a user is allowed to perform a particular action on a resource.

Authentication is the mechanism with which the user can prove who he/she claims they are. A system with a *single sign-in (login)* enables grid users to enter their login details once. These details are used for all resources accessed by the user, instead of having to enter credentials for each resource used separately. The *lifespan* of credentials becomes important when running long-running jobs. There should be a mechanism to notify user if the job is taking longer than the credentials lifespan; the user should then be given the

option of prolonging the life span of the credentials.

Grids should not be tied to one particular implementation of authentication technology; there must be an ability to easily use different implementations. One way of verifying a user is who he/she claims they are is to use *X509 certificates* (see [25] and [26]) The *logging* of user activity is required for audit purposes; the implementation of this should be eased by single sign on.

Trust Relationships

Trust relationships are needed in grid services because they may operate across multiple domains in the course of their work. Entities in these different domains must be able to obtain authorisation information. This is made more difficult by the dynamic nature of grids, meaning trust relationships may have to be found at runtime. Users can also form transient grid services; these services are responsible for completing the tasks set by the user. These tasks may involve using many different resources.

The challenges fall into categories of identity and authorisation, enforcement, deployment, policy and delegation. *Identity, authorisation* issues occur because services have an identity and various authorisations associated with them. It must be possible to set these. The *enforcement* of policies for the transient polices must be implemented.

Assuring the *security of the deployment platform* means having the ability to discover the strength of the platform, for example the state of anti-virus and fire-walling software. Then decisions can be made based on this information. Third party verification of the information used to make these decisions is desirable. The result is that grid users running jobs on remote machines using confidential data can be satisfied that the confidentiality of the data will not be breached.

Policies can be generated at runtime from both the user's transient service and the resource owners. *Delegation* means the service must be able to act on the user's behalf. A service running remotely might require access to another

resource. The service must be able to act, without direct intervention from the user, using the user's credentials.

2.5.3 Resource Layer

The resource layer deals with individual resources, as opposed to the collective layer (see 2.5.4), which deals with groups of resources. Building on the protocols defined in the connectivity layer, the resource layer defines APIs and protocols for the following:-

Single Resource Negotiation is the process of negotiating use of the resource, which is dependent on authorisation, in a secure manner. *Initiating the resource* is the task of doing whatever needs to be performed on the resource before it can be used. Once the resource is in use *monitoring* allows the monitoring of the resource and its activities. *Control* is the ability to control a resource over the grid. Finally, *accounting and payment* is the process of recording what has been done using a particular resource and charging any charging model for the resource.

Resource layer protocols can be divided into two types:- firstly, *information* protocols are used to retrieve information about the resources state, structure and capabilities. Secondly, *management* protocols are used to control access to the resource. The layer enables sharing of the resource consistent with a pre-defined policy. The management must ensure usage of the resource is not in breach of the policy under which the resource is usable.

It is deemed that the number of resource layer protocols should be kept minimal and that they should be tightly focused. At the same time the ability for higher-level layers to construct protocols using resources should not be limited. The protocols must be general in nature allowing their re-use.

2.5.4 Collective Layer

The collective layer builds on the resource layer. Instead of dealing with individual resources, the collective layer works with collections of resources.

It provides APIs (*Application Programming Interfaces*) and services that allow, by being more globally-orientated, interactions across the collections of resources. The collection layer protocols can range from being general to domain or application specific. Services can be implemented as persistent services or as APIs designed to be used in applications. Examples of resource layer protocols are *Directory Services*, which allow VO users to discover resources, *Scheduling Services* which allow VO users to request resources for tasks, *Monitoring Services* that allow the monitoring of resources and *Replication Services* which provide replication of data to maximise the availability of the data. Having multiple replicas reduces access times for frequently accessed data sets.

Typically the *collective layer*s implementation would be in *middleware.*

2.5.5 Application Layer

The top layer, the application layer, consists of user applications that run in the context of a VO environment. Applications are constructed by writing user code and also calling upon the services provided by the lower layers of the architecture to grid-enable the application.

2.6 Summary

In this chapter the need for grids, the architecture of grids, the technologies used to implement them and how they work are all discussed briefly. The way the grid architecture is split into five distinct layers is also introduced.

The chapter should give the reader some insight into why grids are becoming increasing important in large-scale computing.

Chapter 3

General Computation

3.1 Introduction

Development of the Spitfire browser and the DUCK system relied on a practical instantiation of general computational techniques. Here we describe the underpinnings of this development work.

The principles discussed here are more general than grid, in so far as many applications could in principle, and are, developed and deployed outside a grid framework. A description of them is needed to clarify them as they are used in the research discussed in this thesis. For both projects commodity hardware was used as a platform to develop and deploy. The Java programming language was the programming language used with ANT as the build and deployment tool.

3.2 Hardware

Hardware is a term to describe the physical components of a computer. The modern *personal computer* (PC) is built around a *Central Processor unit* (CPU), which is a silicon based semi-conductor microchip. This CPU is mounted onto a circuit board known as the *mainboard* or *motherboard.* Plugged

into this mainboard are card based devices that provide interfaces to display units, storage, networks, scanner, printers and many other items. All the above items are considered as hardware and are described in Appendix F.

3.3 Software

Computers are incredibly powerful machines, but without a means of giving them something to do, some instructions, they are useless. Instructions for computers are generally grouped together as programs. These programs are not unlike a recipe for a chef; there are a number of steps that need performing to get a result.

Like a spoken language, programming languages have evolved and changed through use. The major features added to programming languages have been introduced to combat the problems that programmers have encounters whilst programming. There has been a tendency to standardise languages, but as with any standard, proprietary additions are often added. This can occur because a vendor wishes to introduce functionality not described in the standard, for example. Modern programming languages have evolved to have english-like syntax; making them easier to read for (English speaking) humans.

As programs have increased in terms of complexity, the ease of maintainability has become a prime concern. Maintaining and updating programs is an important part of the software-lifecycle. Programming languages have moved to ease this with concepts such as encapsulation, good documentation (or rather the means for) and Object Orientation. The idea of breaking programs up into to manageable chunks and being able to write down what each chunk does are massive aids to maintaining programs.

There are a number of potentially challenging concepts introduced with Object Orientated programming, but it should be remembered these are just ways to ease maintainability and re-use of code. Both things that save work and increase productivity. In the following section we look at the progression of computing languages.

3.3.1 Assembly

To begin with programmers used machine code, programs were written in terms of instructions to the Central Processor Unit (CPU). This method of programming was very labour intensive, required detailed knowledge of the computer hardwares architecture and meant programs were restricted the architecture they were written for. Machine code evolved into assembly level programming. This meant groups of machine code instructions were 'assembled' into higher-level instructions leading to the ability to make abstractions. The idea of abstractions leads us onto procedural languages.

3.3.2 Procedural

Procedural languages move away from assembly. Instead the abstractions of functions and data-types mean programs can be written without knowledge of the underlying hardware (CPU) instructions.

Here we encounter the concept of a 'black-box' meaning the programmer can call a function using the parameters and not have to worry about what the functions is doing internally. Development for procedural languages is often considered to be structured development; 100% of the programs function is planned before any code is implemented.

3.3.3 Object Oriented

Object Oriented programming originates (like a number of computing innovations) from the Xerox Parc labs. Object Oriented programming revolves around encapsulating data and functions in a package typically called an `Object`. The object tends to represent an entity, often a real world entity, such as a person or car.

Data is encapsulated in the object, often in a form that means it is inaccessible from outside the object directly. Instead methods or functions are provided to access the encapsulated data. These methods or functions also allow the data

to be manipulated. Objects, therefore, combine function and state (the values of the data encapsulated in the object).

Object Oriented programming also introduces the idea of inheritance. Objects can be extended to create new objects that inherit the properties and functionality of their parents. Using this model a generic object, say a person, could be programmed. It could have the attributes name, date of birth, address and so on. Functionality is then provided to get the values of these attributes; for example to calculate and return the person's age. The object could be extended to create the object customer, which could have the additional attributes of delivery address, the products they have ordered, etc. Functionality to access these attributes and extra functionality to perform operations to this data could also be added.

Object Oriented programming aims towards the programming holy grail of reusability. Allowing programmers to engineer components that can be reused in many software projects.

So in a very rough summary code is contained in methods. Methods act on and are contained in classes. These classes form the building blocks for programs.

3.3.4 Namespaces

Languages such as Java[27] and C++[28] have the concept of namespaces. In Java namespaces are called packages; but the concept is the same.

They are a mechanism to resolve ambiguity with respect to naming in programming. For example you might write a class called Mouse to decode the movements of an input device, a mouse. Someone else could also write a class called Mouse, perhaps to simulate the activity of a mammal mouse. If the two classes are used in one program there needs to be a way to distinguish between them. Using namespaces, we have a mechanism to distinguish between the two classes. For the input device we could have the namespace 'input' and for the animal we could have the namespace 'biosim'. In Java we could then use the prefix `biosim.Mouse` and `input.Mouse`; they would then reside in the

packages biosim and input. Of course to use the classes we would need to import them into the Java program we were writing.

The names within a single namespace/package should be unique. A packaging structure also allows different components (classes) of the software to be arranged in logical groupings. For example, we might want to group all the classes related to controlling the mouse in to one package.

3.3.5 Threading

Threads allow programs to do many tasks at once. Programs can be written to only have one thread of execution, but this severely limits program performance for some applications.

As an example a program might provide a User Interface to interact with the user. If the user initiates a task that might take a long time to complete, the display should still be updated (ideally with the current progress of the task). To do this two threads of execution are needed; - i) To carry out the task ii) To update the User Interface

Most modern programming languages have support for threads. The use threads introduces some issues in programming that need to be addressed. Synchronisation between threads is an important issue; threads need to be able to interact with each other and share data. If a number of variables or objects need updating by a thread, a lock is used to prevent another thread either reading an incomplete set of data or writing data before the other thread has finished.

In situations where a thread only does a limited amount of work periodically (for example, checking for new email every 5 minutes) then the thread will yield to other threads and sleep, allowing the compute cycles to be used by other threads.

3.3.6 Programming Platforms

The Java[27] programming language started life as a language called Oak in the early 1990's. Oak was just one part of Sun Microsystems Green Project (see [29]), conducting blue sky research to investigate future consumer computing devices.

After its release Java quickly gained developer popularity. The language is a modern language, which borrows from modern Object-Oriented languages. It discards some potentially confusing and troublesome features such as multiple inheritance.

A core goal of Java is Machine Independence; the write once, run anywhere concept. Java code is written for a Java Virtual Machine (JVM), a piece of software that is responsible for executing a Java program. Hence, a Java program should run on any machine that has a JVM written for it. Java programs can take the following forms: -

- Applications, a Java program running as an application, interactively with the user, possibly with a user interface and full access to resources such as files, network sockets, printers etc.

- Applets, a Java program which runs in a *sandbox* and appears to be part of the web page being browsed. Applets are seamlessly downloaded from web sites and tend to have restricted security models to prevent the problems of malicious code.

- Servlets, Java programs which run inside a *servlet container*, see section 5.5.3. They handle requests from clients. Quite often servlets are HTTP servlets, meaning they are accessed over HTTP (like a web page) and the response of the servlets actions is a web page.

- Java Server Pages (JSP) are snippets of Java code are embedded inside HTML code (the code that builds up the web page). JSPs are looked at in more detail further on in this section. ASP (Active Server Pages) are an alternative technology from Microsoft.

- Java code can be embedded inside databases as stored procedures.

There a three flavours of Java; the Standard edition[27] - aimed at running Java applications on desktop computers, the Enterprise edition,[30] - aimed at running web applications on servers and the Micro edition- aimed at running applications on small devices such as Personal Digital Assistants (PDA). The standard edition (versions of 1.4.x) were used as the platform for developing both Spitfire (see section 6) and DUCK (see section 8).

Java has always been a network-oriented language; providing easy access to sockets (see 5.2.2) and web-orientated features. This lead to Java technology being adopted by Netscape for their browsers. This lead to applets becoming relatively widespread.

Java also has a wide variety of packages to provide the programmer access to databases, Image and media processing, user interfaces, distributed computing, native code, XML and many other features. Aside from the a few primitive types, all objects in the Java language are rooted from the *Object* class. This means every object has the functionality provided by this base class.

Instead of allowing multiple inheritance Java provides interfaces. Interfaces are like having a class with no implementation; only the method signatures are provided. In Java a class can only extend one class (inheritance) but can implement many interfaces. For each interface the class has to provide an implementation of the interface; filling out the methods signatures with an implementation that does the work.

Java has the significant advantage of having *automatic garbage collection.* This means when objects go out of scope (meaning they would not be used again) they are garbage collected by the virtual machine. In comparison, in languages like C++, the programmer is responsible for garbage collection; any memory allocated by the programmer must be freed when the programmer is finished using it. This requirement often leads to memory leaks and program errors.

3.3.7 Build Systems

The development process involves writing, compiling, running code. Often, especially when debugging, this process may need doing many times. Manually using a compiler and then performing various other actions may become labour intensive. The program may need complex arguments to compile, or have many source files, or sometimes only one file will need re-compiling.

The solution is to have the instructions for making or building the program in a form such that they can be re-used easily.

This is the concept of Make[31] and more specifically a 'Makefile'. The makefile contains instructions on how to build a program. Make can also perform other actions by having instructions grouped into targets. Each target can do a particular set of tasks; for example one target might build the program, another might install the program and a third target could be used to build the program with extra debugging information.

Of course shell scripts could do much of the above, but are platform dependent. Secondly make systems 'know' which source code has been modified. This means the make system can selectively compile, compiling only the files which have changed. For a large project this can save a huge amount of compile time.

Another Neat Tool (ANT), part of the Apache project, see [32]) is similar to GNU Make in terms of its functionality.

The major difference with ANT is that the makefile is an XML file. It is also Java based and, by Java's nature, platform independent. The build files ANT uses disposes with the need for the pedantic syntax of the Makefile.

ANT is widely used in the Java community and is supported by many of the IDE's (Netbeans [33] being one example)

3.4 Summary

In this chapter the nomenclature and concepts of general computing have been introduced. These concepts are more general than grid computing which was considered earlier in this thesis. Furthermore, these concepts provide the foundation for the later chapters in this thesis, which consider the implementation of the Spitfire (see chapter 6) and DUCK (see chapter 8) projects.

Chapter 4

Data, Structure and Storage

4.1 Introduction

In this chapter the concepts and ideas surrounding data; the structuring of data and its storage are discussed. Persistent data storage is approached from two angles; firstly, the structuring of data for storage and, secondly, the technologies used to store the data. The discussion of data and metadata provides background for my Spitfire project-based work (see chapter 6). The structuring of data, namely in the form of XML, and the storage of data, particularly in relational databases gives some of the basics on which my MAP project work is built.

4.2 Data

Data can take the form of text, numbers, characters, images and a large number of other formats. The term tends to refer to entities that are stored, processed or transmitted, often using computer systems.

Information is said to be created when data is processed. The process of 'spotting patterns' in data can be done by humans or by computer programs; this is common form of processing.

4.3 Volatile Data

When a program finishes running the data stored in its memory space is usually cleared out or lost. In addition, modern languages also have a feature that is referred to as *garbage collection*; meaning that data is cleared when it is no longer referred to by the program. Furthermore, the contents of *Random Access Memory* (RAM) does not survive when a computer is powered off.

4.3.1 Variables, Memory Structures

Normally volatile data in computer programs is stored as variables. These variables have *types*, for example, int or integer for storing whole numbers and char or character for storing characters.

Areas of memory can be assigned in blocks to store larger amounts of data, for example an image is represented by a series of bytes, describing the colour value of each pixel. In cases where large blocks of memory are assigned two methods can be employed. Traditionally, programmers had to allocate blocks of memory, use them and finally signal to release them when he/she has finished using them. Failure to release memory blocks leads to memory leaks. Secondly, there are modern programming languages, which garbage collect automatically; in this case the programmer need not worry about remembering to release memory blocks.

Data can also be encapsulated in classes (see object orientation, page 19) enabling data fields to be accessed via 'instance methods'. Methods are typically small pieces of code that allow data in and out of a class; ideally there is no direct access to the private class members. This allows the construction of complex data structures; more commonly known as *abstract data types*. Classes are similar to structures, however structures lack instance methods.

Given that classes can also have complex interactions with other classes, saving them to disk can be a complicated procedure. This process is generally known as serialisation and is discussed in section 4.4.1.

4.4 Persistent Data Storage

4.4.1 Introduction

As discussed in section 4.3, there is a need to maintain data between program executions and indeed when data needs maintaining in stateless protocols. We will discuss this aspect of persistency in DUCK, see section 8.3.4.

Therefore, data needs to be written to a persistent media. Examples of which are magnetic media, such as hard disks, tapes and floppy disks or optical media, for example, CD-ROMs, DVDs and laser disks.

Retaining Data

Persistent data can be user data (for example, documents; images, text, numerical and data), or settings (for example the placement of a window in a graphical environment). This data tends to be stored in either files or a database system.

The process of storing the data is referred to as *serialisation.* This term is used especially when talking about object instances being written to disk.

Serialising Data

Frequently in object orientated languages the language has facilities to serialise objects to disk. This means the object is written as a stream of data into a file. The object can then be 'recreated' from the file at a later time. If the language does not support serialisation; then object databases (see section 4.7.2) provide this functionality.

Serialising instated objects can be a complex process; due to the need to store not just the objects but to also map the inter-relations between these objects. This problem is discussed at length in [34].

The use of serialisation in programming to provide persistency can lead to

problems, for example different versions of a platform can produce incompatible ways of serialisation.

Programs can also, and more often do, provide persistency by writing out a file to a particular format- which may not bear any relation to how the data is stored in memory. These file formats are often proprietary, making it difficult to write other programs to process them. This problem is addressed by *eXtensible markup language* (XML), which is discussed in 4.6.1 onwards.

4.4.2 Files

Flat files are possibly the simplest form of storing data on a computer. They can contain data such as the text of a word processed document, an image or raw data. Frequently data is stored in proprietary formats, this makes the exchange of data between programs difficult. Data stored on desktop computers is mainly in the form of flat files, arranged in a hierarchical structure. Whilst various operating system vendors have talked about using databases as file systems, it has yet to happen.

For an application where there are many concurrent users (or processes) that use a single flat file problems can occur; with a major one being corruption of data. Often there is no locking mechanisms on flat files, which can lead to the files being updated incorrectly if two or more users are making changes. Even with locking mechanisms race conditions can occur, this is where two processes fight for the lock on a file, the result of which being that the file can be corrupted or even wiped.

The use of flat files leads to problems relating to access. If many users need to access a single file on a desktop computer (perhaps over a work group network) then bottlenecks will occur.

4.5 Metadata

Data on their own can lack meaning. Think of a photograph in a dusty photo album; without a written caption or a few words of explanation, it could become meaningless after the events it depicted have been forgotten. Similar problems exist in computing; meaning, explanation and added data needs be associated with data. Hence, we have a requirement for *metadata.*

Most definitions of *metadata* define it as 'data about data'. Therefore it could be described as a subset of information that describes the data that the metadata is associated with.

Metadata is important in modern distributed systems; one use of metadata is to describe resources in a way that allows systems to interact with each other automatically.

Metadata also enables lineage data to be collected. Such data could, for example, be used to keep track of users changes to a particular document. An *audit trail* could then be done using the lineage metadata; this would detail all the changes made to the document; what was changed, who made the changes and when they were made.

Another use of metadata is to describe resources in terms of the information that they contain. The *Dublin Core Metadata Initiative* (DCMI)[35] defines a set of metadata elements that are commonly used to keyword web pages. As a result humans and automated search engines can categorise pages.

The concept of ontologies is related to metadata. Ontologies provide a way of specifying how knowledge and information of a particular domain of interest are mapped into some structure. Typically this is done to aid automated processing of the information.

4.5.1 Types of Metadata

Metadata can be classified into two types, *immutable metadata* and *independent metadata.*

Immutable metadata means that the metadata cannot change without the data itself changing. An example of this is the file size; the file size can only change if the amount of data in that file changes, this in turn means the data will have changed.

Independent metadata can change without the data changing. An example of this is the filename; a file can be renamed without changing the data contained in the file.

Metadata can also be classified into the following categories:-

- *Technical metadata.* Technical details, for example how the document should be printed.
- *Administrative metadata*, an example of which is lineage metadata; this can record the actions that have been performed on a document by its editors. The metadata can be reconstructed to form an *audit trail.*
- *Structural metadata* describes how a resource is structured. It records how the document is constructed from its compound elements. For example a document may be structured into logical sections.
- *Descriptive metadata* provides a description of a resource; typically these metadata elements are keywords. Associating keywords with a resource allows the discovery and identification of resources. Elements in the DCMI are a good example of descriptive metadata, see appendix B.

The Spitfire project (see section 6) dealt with most categories of metadata. DUCK (see section 8) used technical metadata to describe details for jobs, administrative metadata to log user sessions and descriptive metadata to provide feedback for error situations.

4.6 Markup Languages

4.6.1 Introduction

Markup languages are a concept that allows data to be marked-up with extra information. Such information can be the metadata discussed above.

The concept of markup languages in computing is a powerful yet simple concept. Extra information is often required in written language and publishing; this must be included somehow in the text. The marking up of texts allows this information to be included in a written format. Elements such as **bold type face**, footnotes, underline and section headings had to be 'invented'; as they simply do not exist in spoken language.

It would seem that the marking up of texts is something that has been done for a long time; there is evidence of medieval texts having markup in the margins. In modern times computing markup tends to be used to provide information to the machine. Hence, it has to be specific and consistent; a computer requires specific commands, it cannot guess what is meant by something that is ambiguous.

There exists a great number of markup languages, examples of which include SGML, HTML and XML all discussed below. SGML tends to be used in publishing and pre-dates HMTL, which is widely used as the language of the world wide web. HTML is mostly concerned with presentation; XML is a more modern technology that aims to address issues in the field of semantics. From an XML viewpoint the meaning of content is a more important issue than the presentation of the content. The world wide web consortium (W3C) is responsible for developing and over-seeing much of the technologies described in this section.

XML was used in DUCK to provide a settings system and a method to define queries in the database system. SOAP, an XML based communications protocol, was the intended transport medium for the Spitfire project.

4.6.2 SGML

The motivation for electronic markup languages initially came from typesetting demands. With the 1960's came the first attempts to produce open and standard markup forms, allowing the transfer of documents between platforms. GenCode and General Markup Language (GML) were the first two attempts.

GML was conceived by Charles Goldfarb and his colleagues at IBM, as a format primarily used for legal documents. Under the auspices of the American National Standards Institute (ANSI) Goldfarb and his team worked to create a more general language, Standard Generalized Markup Language (SGML). It was ratified as an International Standards Organisation (ISO) standard.

SGML is a meta language, meaning it can be used to define other markup languages. To this ends, it is extremely flexible and has a very generic coding scheme. Due to the complexity of SGML, the tools for processing it tend to be complex and expensive. The software tools used to process markup documents are known as parsers. These read the document from a disk and turns it into a series of tokens. These tokens are typically used to create either a model of the document in memory or generate a series of events that can be interpreted. Parsers are discussed in greater in section 4.6.5.

SGML has attained a great deal of success in publishing and large industries, where the expense and complexity of the software tools are affordable.

4.6.3 HTML

Developed at CERN by Tim Berners-Lee (and influenced by Anders Berglund) as language for publishing content on their world wide web, *HyperText Markup Language* (HTML) has proved to be a global success story, which has been instrumental in the expansion of the world wide web. HTML is a language defined using SGML. It defines a set of tags that are more concerned with the presentation of content on the web than anything else.

One of the most fundamental elements in HTML is the *anchor* element with its

href attribute, allowing hyperlinks to be created. A hyperlink may be created to another page, image or other such resource. When the page is rendered for the user the hyperlink appears as an underlined piece of text. The user can then click on the link and is taken to the corresponding resource or page.

Processing software for HTML tends to be reasonably forgiving about inconsistencies and mistakes in the markup, meaning that whilst it makes HTML perhaps more accessible, the processing software is a good deal more complex. This problem is compounded by the number of proprietary tags that have been introduced in the 'browser wars'[1]. HTML only defines one document type; an HTML document.

4.6.4 XML

eXtensible Markup Language[36], [37], is the human-readable form of the acronym XML. Like SGML, XML is a meta-language, allowing it to define other markup languages, development of the language was spurred by the W3C consortium.

XML was partly created to address the short-comings of HTML. Due to it is forgiving nature to markup errors, writing web pages is less demanding than, say, programming; less 'strictness' has made it 'easier' to produce web pages. This is turn, has led to HTML parsers having to be able to understand a whole myriad of technically incorrect HTML.

XML was created to address this problem; by having stricter rules for the well-formedness of markup. HTML markup predominately describes presentation, meaning that the function of the markup is to describe how the content should be displayed to the user. Increasingly, there is a need to describe the meaning of the content; this is *semantics*[2]. A example of this is an address:-

[1]The 'browser wars' were a succession of releases of versions of the Microsoft Internet Explorer browser, the Netscape browser, the Mozilla browser and, at a later date, the Firefox browser.

[2]*Semantics* refers to the meaning of some element in a language, as opposed to syntax; which describes the combination of elements. The idea of semantics is important in com-

D Jones
Oliver Lodge Laboratory
The University of Liverpool
Oxford Street
Liverpool
L69 7ZE
United Kingdom

A human, with some experience of UK addresses, can make sense of the address and attach meaning to the various elements of the address. A computer just understands this as a block of text. One can use semantic markup to add some meaning:-

$$\langle person\rangle HerMajestytheQueen$$

$$\langle address\rangle$$

$$\langle dwelling\rangle BuckinghamPalace\langle /dwelling\rangle\langle street\rangle TheMall\langle /street\rangle$$

$$\langle city\rangle London\langle /city\rangle$$

$$\langle postcode\rangle SW1A1AA\langle /postcode\rangle$$

$$\langle country\rangle UnitedKingdom\langle /country\rangle$$

$$\langle /address\rangle$$

$$\langle /person\rangle$$

This example is a valid piece of XML. Now, with the correct instructions, a computer can extract some meaning from the fragment. The markup has been used to introduce meaning into the information. Also inferred by the nesting of elements is a hierarchical structure. This allows XML to store data in a hierarchical fashion. XML also signifies a move towards a model in which data is in an XML file and the presentation is handled separately.

There are also a large possible number of different ways to markup the address; differing amounts information could be used, it could be structured differently

puting, as attaching meaning to data allows computing to have an 'understanding' of data and to perform meaningful operations on it.

or different element names could be used. This provides the motivation for having *schemas*, which are covered in appendix C.2. The motivation for moving away from the 'forgiving-ness' of HTML was to simplify the writing of processing software. HTML parsers have to allow for many different 'inconsistencies' in markup; whereas XML is stricter with the well formedness of markup. This leads to parsers being simpler to write, less complex and smaller. This is important, as several dialects of XML are intended for portable devices with less processing power and storage; Wireless Markup Language (WML) is one example.

The web is not the only area to have problems that XML aims to solve; applications are also a potential beneficiary. Traditionally, applications have tended towards the storage of documents in proprietary document formats. The documents are written in such a way and format that is only understood (and often carefully guarded) by the company writing the software. There is a potential to extend XML to create open and interchangeable document formats. Another application of XML is the exchange of data between businesses and over the network. Here a cross-platform and open approach is desired. Finally, mundane issues such as persistent settings for programs (for example where the application window is displayed on the screen) need to be stored, with XML being another potential provider.

The web application DUCK (see section 8) uses XML to store settings information such as addresses, database locations, custom queries and drivers, and some application-specific configuration. In addition the servlet container (see section 5.5.3) uses XML to store its configuration.

Several of the features of XML have been touched upon in section 4.6.4. A basic, but important, feature of XML is that the standard character set is unicode, it can, therefore, support characters from a huge number of international character sets. Like HTML, XML uses tags to markup content, these tags must be such that the resulting document is well formed. The tag names can be defined by the creator of the document, meaning that new markup languages can be created using XML; XML is a meta language.

The same markup tags can be nested to represent a hierarchical structure in a document. XML start tags must have matching end tags, this is a departure from HTML. The exception is if the start tag self-closes. This requirement is due to the fact that the XML parser may not have knowledge of the intended structure of the document and would run into trouble deciphering between sibling elements and child elements. Forcing every tag to have a matching closing tag means that the structure of the document is unambiguous.

Attributes can be added to elements. Again XML is strict with the well formedness of these. Attributes must be correctly quoted with quotes at the start of the value and at the end of the value.

It should be noted that XML does not replace HTML. In the short term the technologies will sit side by side; XML for data with meaning and HTML for presentation (perhaps generated from the XML). In the long term the HTML reformulated using XML; XHTML will be increasingly used.

Many, or perhaps even most, projects that would normally use SGML are now using XML instead.

4.6.5 Application Programming Interfaces

XML *Application Programming Interfaces* (APIs) allow programs to leverage the data stored in XML documents. Typically parsers are used to process XML documents; they are libraries of methods and mean that software developers do not have to constantly write methods to process XML from scratch.

The concepts of a parser were introduced in section 4.6.2. An *XML parser* is a markup processor that turns the XML located in a file into *tokens* or fragments of text. These tokens can either be used to trigger events in the program or to build up a memory representation of the data. Parsers also provide ways to serialise the document, ready to go back into a file.

The two main ways of using XML; *event based parsing* (typically SAX) and *Document Object Model* are described in more detail below. Here we will

consider the parser mainly from a Java point of view. Bear in mind that XML is supported (via libraries) by many languages including C/C++, Perl, Python and C sharp.

The *Document Object Model* (DOM) is a W3C recommendation and is an alternative to using SAXs event based API. Instead of generating events for each element, a DOM parser creates a model of all the data in the file in memory. This means the entire document is available to the program concurrently.

The DOM model itself is standardised. The set of interfaces to access the document elements is a standardised API. The parsing can be accomplished by a number of different parsers, providing the chosen parser is compatible with JAXP (Java API for XML Processing, see [38]).

The interface to the DOM was designed to be as language neutral as possible. It therefore, does not use Javas collections classes, for example *vectors* and *hashtables*. Java programmers will be accustomed to being able to use these objects.

Instead, the document hierarchy is transversed using the classes `Node`, `NodeList` and `NamedNodeMap`. Most classes in the DOM are subclasses of the `Node` class. `NodeList` encapsulates the child elements of a `Node` and `NamedNodeMap` provides the equivalent of hashtable functionality (value lookup by hash key) for child nodes.

The programming model provided by DOM is easier to work with than SAX for complex documents. The downside, due to the entire document needing to be in memory, is high memory usage for large documents. DOM was chosen as the XML interface to DUCK; it provided the system to read in application specific settings and the custom SQL queries.

4.7 Databases

4.7.1 Introduction

In this section databases are discussed. They can be considered as collections of related, structured data. This data can take many different forms, from names and addresses of customers, to component listings for a particle detector or a regional re-stocking list for a supermarket chain.

The *relational database* takes its name from the theories it is built on; *relational algebra* and *relational tables.* It is the most successful type of database, partly due to *Structured Query Language* (SQL), we will look briefly at this language in this section, which is also a data definition language.

Typically, motivations for databases include:-

- Large data sets.
- Being able to handle concurrent user access, either many users reading the database or managing many users updating the database.
- Centralising data to simplify management, reduce duplication of data, manpower and resources.
- Giving different views of the same data, either for analysis or to limit the amount of data visible to users.
- Transactional processing; the ability to group operations into a transaction. If any one of the actions fail the transaction is failed and the database is left unchanged. If all the actions in the transaction complete successfully then the transaction is committed to the database.

Database Management Systems (DBMS) are a collection of programs used to manage a database. The combination of the DBMS and databases is usually referred to as a *database system.* Typically, DBMSs possess the following functionality:-

- Restricts access to data. Users may (and often do) require varying levels of access to data; some users may have read-write access whilst others only read access. Read access to certain sensitive data may be restricted to a subset of users.

- Provides data integrity checking via integrity constraints. The most simple form of integrity constraint is to specify a data type; much like in a programming language. More complex constraints tend to involve the semantics of data; somewhat like the schemas described in section C.2. An example (in, perhaps, a product shipping database) of this would be 'every order must have a related shipping address'.

- Provides backup and recovery. In the event of hardware or software failure, recovery of the database should be possible.

Databases and their use at CERN are described in detail in [39].

4.7.2 Types of Database

A description of a database type can be divided into 3 pieces; the *data model*, how the data is stored, the *Query/ Data Definition Language* and the *Computational Model*, how data is accessed and processed. Databases can be divided up into types by the model they use to store data; the data model. The data model theory has been an important part of the evolution of databases, examples of database models are hierarchical databases, network databases, relational databases, object databases and object-relational databases. Up until the 1980's the most common data models were hierarchical and network data models, subsequently, the relational database model rapidly became the data model of choice. The relational database is used to provide database services for projects described in this thesis.

Hierarchical Databases

The concept of a hierarchy will be familiar to some computer users as the way in which Windows Explorer displays directory structures. The hierarchical model can also be described as an upside down tree. The database has a root with children branching out from it. Children are linked to parents using pointers or tree paths. Whilst the hierarchical model is more efficient than flat files, there are a number of drawbacks:-

- Entities cannot be added to the system unless they fit into the hierarchy. For example if you structured a database around products and then had customers as children of these particular products, you could not add a potential customer to the database as it stands.
- Duplication of data is undesirable; take the example of a customer buying two products, then the customers would be added twice under each product.
- In short, hierarchical databases handle one to many relationships, but not many to many relationships.

Network Databases

The *network database* attempts to address the limitations of hierarchical databases. Instead of using hierarchies the network model uses *sets* to represent relationships, therefore, children can have more than one parent, which somewhat eases the problems of hierarchical models. The network model can still be viewed as a tree or more specifically a set of trees that share branches. There are also disadvantages to the network models, namely difficulties in implementation and maintenance, resulting in the network model being used more by programmers than end users.

Relational Databases

Relational databases have dominated the database market since their arrival on the database scene, these are discussed below. The Spitfire project (see chapter 6) was conceived to allow metadata storage in relational databases. DUCK (see chapter 8) uses relational databases for storing persistent data.

In relational database theory *normal forms* provide a set of guide lines, which prevent data duplication in relational database tables. This helps to reduce inconsistencies in data and anomalies when updates are made to the database tables.

There are several normal forms or rules, which are numbered sequentially. The first normal form deals with the number of fields in each record. It states that there must be the same number of fields in each record; which is always the case in relational tables.

The second and third normal forms deal with the relationships between key fields and non-key fields. The second normal form deals with subsets; it eliminates functional dependencies (the definition of functional is quite close to the mathematical definition). For example, say you had a table with data on people renting houses. If this table contained the address of the house, as well as the clients renting, the second normal form says this should be moved to a separate table.

The fourth and fifth forms deal with multi-values, where there are many to many relationships.

Object Databases

Whilst relational databases rule most of the database market they are not ideal for certain niche areas of data storage. Applications such as engineering computer aided design, software engineering, scientific experiments and multimedia (for example, videos, audio and images), have demands that are not best handled in a relational database.

These types of application tend to have demands for new data types (such as images), more complex data structures (such as Objects), the requirement for operations on data, which are demanding on the application and much longer in durations of the transaction.

To fulfill some of these requirements the *object database model* is used. These object databases can model the increased complexity of the above applications.

Object databases are closely linked to *object orientated programming languages* (See section 3.3.3). The database can provide bindings, allowing access to the database from an object orientated language seamlessly. This means the database almost becomes an extension of the language, providing persistency easily to the application. Such an ability is important when object databases are used in software development projects.

Relational database vendors have responded to object databases by developing extensions for their databases to provide some of the object functionality in their databases. Such databases are known as *object relational databases.* A discussion of object management is given in [40].

Object-Relational Databases

Object relational database systems are an attempt by the relational database vendors to add object-like functionality to their existing relational database systems. The SQL language has been extended to to provide object query capabilities. The ISO standard ISO/IEC 9075-10:2003, SQL Part 10: Object Language Bindings (SQL/OLB) provides a standardised specification, see [41].

4.7.3 Advanced Database features

4.7.4 Database Connectivity

Database connectivity libraries/APIs provide programmers with a standardised way to access databases, therefore, clients can programmatically access

databases using the provided libraries. The two packages commonly used are *Open DataBase Connectivity* (ODBC) and JDBC (which is not actually an acronym, but is commonly understood to mean Java DataBase Connectivity, see [42]). JDBC is used to provide database connectivity in Spitfire (see section 6) and DUCK (see section 8).

Programming API bindings

ODBC is based on SQL Call Level Interface (SQL CLI). In turn JDBC borrows heavily from ODBC and hence SQL CLI. ODBC interfaces provide consistent programming interfaces for different data sources.

The data sources are usually SQL databases, which differ in implementation between vendors. The client application will see just one interface (SQL CLI, ODBC or JDBC) that provides access to the data source. All three systems pass SQL queries to the data source, of course vendor implementations of SQL differ, so often the libraries have to allow for this. In the core software there are classes that allow queries to be sent, results to be manipulated and resultant metadata to be inspected. Differences between the data sources are dealt with using modules known as drivers or database drivers. Each vendor will release a driver for their database product. As a whole (combined with the driver) the libraries will deal with connections to the database, the submission of queries and handle the results.

ODBC

ODBC was developed by Microsoft, but is an open standard. It is designed to be language neutral, however in practice tends to be orientated towards the C language. One criticism of the ODBC API is that it is fairly complex, even when doing simple tasks with the API. Microsoft ships Windows with ODBC support. ODBC has been ported to other operating systems including UNIX. See [43] and [44])

JDBC

JDBC was developed by Sun Microsystems as part of Java and hence is Java based. It was designed to address what Sun saw as the short-coming of ODBC. The ODBC API was unsuitable for use with Java because it makes extensive use of pointers (memory location variables), a concept that does not exist in Java. Also, Sun wanted to provide an easy to use API, but also allowing more complex tasks to be done.

JDBC has a driver called an JDBC-ODBC bridge; this allows ODBC data sources to be used using JDBC and Java. The API for JDBC is given in the package `java.sql`. The package `javax.sql` provides an API for server side processing of data sources. In the package `java.sql` the important parts of the API are;-

- `Connection`. This interface represents a session with a specific database. It enables statements to be created that allow SQL statements to be used to query the database.
- `Driver`. This interface specifies the methods that must be implemented in driver classes. The implementing classes provide a method to connect to a specific database using a URL, returning a connection object.
- `ResultSet` Interface. This represents a table of results, in which the data is constructed in rows, from a database query. The object maintains a cursor pointing at the current row. Using the method *next()* the next row can be obtained, and *getXXX()* methods are used to provide access to the fields (or columns) in the rows.
- `ResultSetMetadata`. This interface provides metadata on the result set it was obtained from.
- `Statement`, an interface that provides methods to execute a static SQL statement an return the results as `ResultSets`.
- `DriverManager`. This class provides services for managing JDBC drivers. The class can either load drivers on initialization or load them on demand

using a classloader. When a connection to a database is requested it is the driver managers job to find the most appropriate driver for the connection.

- `SQLException`. A class providing exception information about any database error or database connection error.

JDBC and ODBC borrow concepts from each other and are inter-related. JDBC driver types fall into the following catergories of type I, type II, type III and type IV (see appendix E).

4.7.5 XML and Databases

When it comes to storing XML in databases there are two main schools of thought; the use of existing relational databases enabled for storing XML or native XML databases. Database vendors such as Oracle, IBM and Sybase have developed extensions to allow XML data to be stored in relational database tables.

A native XML database is designed from the ground up to store XML data. This means there is no need to try and map the data from the XML into some other format. XML databases can be considered as seamless XML storage entities. XML native databases are ideal if your data is XML and you want to store and retrieve it as XML.

There are a number of native XML databases available, many of them under the open source license. The Apache group (authors of the Apache server and Tomcat servlet engine) produce Xindice[45]; a native XML database.

Xindices functionality is accessible via a number of methods. There is a command line tool set, a Java API and an XML RPC plug-in (making Xindice available from languages other than Java). The query language for Xindice is XPath and an update language is provided by XML:DB XUpdate. The Java API is an implementation of the XML:DB API; which is a standardised API for native XML databases. Another way of working with the Xindice database

is to retrieve data and XML documents as DOMs and work with them in that form.

Xindice organises (or rather the user organises) the XML documents in its database into collections. These can be nested within each other; hence a hierarchical system not unlike a UNIX or Windows file system can be built up. Xindice provides the *CollectionManagerService* to allow collections to be added and removed. Version 1.1 has the useful feature of being a web application, meaning it can easily be installed in a servlet engine.

4.8 Summary

In this chapter a number of key concepts that are used to implement, and relate to, the Spitfire project (see section 6) and DUCK (see section 8) are introduced. The importance of metadata is considered, along with the methods that can be used to structure data. Finally, databases are considered with a particular emphasis on relational databases.

Chapter 5

Cluster and Network Computing

Clustered and remote computing are discussed below as an introduction to the large scale fabric used for MAP and other computer clusters. The standard techniques for controlling such clusters are described along with networking and distributed application techniques which are also relevant in the description of Spitfire.

Again a distinction is made between the contents of this section and the description of grid. These techniques and technologies can be applied outside of the grid environment.

5.1 Clusters

A computer cluster is a grouping of computers connected by a local network. They operate together as a unit.

5.1.1 Characteristics of Clusters

Typical motivations for building computer clusters and their resulting characteristics are; -

- To provide *load balancing.* In situations where systems have large loads, such as a web server or database servers, additional machines can be used to spread the load.

- To provide *high availability.* For situations where a system must always be available, for example, if a machine running an e-commerce site fails then customers will not be able to place orders. Having a cluster of machines allows other machines to replace failed machine without interruption to the service.

- To provide *parallel processing.* A cluster of computers can be for running computing jobs in parallel. Typically scientific applications benefit from this approach and quite often clusters are built from commodity hardware. The clusters for this type of application are built by networking a large number of machines together. Computer clusters for computational applications is the area this section looks at in more detail.

Some clusters will run many different kinds of software, for example high energy physics clusters will run jobs with the executable supplied by the user. Some clusters will spend all their time running the same software application, for example aerodynamic simulations.

Many HPC applications required a fourth type of cluster, this is a low cost cluster delivering huge numbers of CPUs often known as a 'farm' or sometimes as a 'dumb' cluster. The individual elements of the cluster know nothing, or very little, of each others existence. MAP is one such example.

5.1.2 Hardware

The fabric discussed here is distinct from that which was discussed in the grids section; the elements here are operated as a cluster and could form a complete grid resource in their own right. In terms of packaging, cluster nodes can be housed in standard tower cases, 'pizza' box style cases (horizontally mounted, shallow boxes in racks) or 'blades' (vertically mounted thin boxes in racks).

Most cluster hardware is commodity hardware. Companies like *HP* and *Linux Networx* build clusters for customers. They tend to integrate commodity hardware to provide their solutions. Having sufficient cooling is an important consideration for clusters; a large number of machines in a single location will generate a large amount of heat. At high temperatures (greater than 40 degrees celsius) the lifespan of the machines (CPU especially) is reduced or for a high enough temperature, failure will occur instantly. Therefore, in addition to the system and processor fans, external methods of cooling are required.

Such methods include air conditioning systems and water cooling systems. In water cooled systems the water is used to exchange the heat from the location of the machines to the external environment.

5.1.3 Cluster Interconnect

Machines forming a cluster need some form of interconnection; this is provided by a network. This network consists of wiring (or transceivers in the case of wireless) and a switching mechanism to route traffic. When large numbers of machines need to be interconnected with switches, the performance of the switches is critical. Therefore, the network for a large cluster tends to be based around high-performance switches with the capacity to handle the network traffic. Companies such as Force10[46] provide high performance switches. The network performance in some computing jobs is critical if there is a lot of interprocess communication.

Many clusters are starting to use gigabit ethernet as the price continues to fall.

5.1.4 Storage

Often there is a requirement on clusters for centralised storage. Having shared storage simplifies storage management. Some examples of storage schemas are:-

- NAS - Network Attached Storage

- SAN - Storage Area Network
- DAS - Direct Attached Storage, storage that is not networked, this term is to differentiate this type of storage from the two above.

SAN accesses data by using a lower level than files; it works at a block level. Many SANs implementations use the Small Computer System Interface (SCSI) communication protocol to access data over the network.

There are various ways of implementing SANs; the most common is using SCSI commands over a fibre channel network. A variation on this still uses the SCSI command set but transmits over a TCP/IP network. This is known as iSCSI.

Finally, ATA Over Ethernet (AOE) uses the Advanced Technology Attachment (ATA) protocol over a raw ethernet protocol.

NAS is a cheaper alternative to SAN. Instead of working with blocks NAS works at the level of files. Examples of the protocols used by NAS are Network File System (NFS) and Server Message Block (SMB). Microsoft reimplemented SMB as Common Internet FileSystem (CIFS) see [47]. The respective merits of NAS and SAN are discussed in this reference [48].

NAS works over a Local Area Network (LAN) instead of having a dedicated network for storage. This reduces costs.

5.1.5 Cluster Management

Fabric management is an important aspect of clustered computing, involving the installation of software and the monitoring of the cluster.

Installation of software on a cluster of many tens of machines requires an automated installation and management system. The man-power costs of installing software on machines on a machine-by-machine basis would become prohibitive for large clusters.

Software is required to manage the submission of jobs to a cluster. Such software might take into account current load on the cluster, estimated job times

and priorities specified by policy. Such software is known as *Job Scheduling* software.

For jobs that are executing the status needs to be monitored and feedback given to either the person responsible for the cluster or the end user. Feedback for why a job has failed is especially important.

When a job has finished running there must be software to manage the output of the job. Such output can be substantial.

On a computing cluster or a batch system a job scheduler provides the following functionality; -

- The abilities to define work flows and dependencies via interfaces
- An automated submission system
- The ability to monitor submitted jobs
- Queues for unrelated jobs submitted to the system that prioritise execution of the jobs

Portable Batch System (PBS) (see [19]) is one example of a job scheduling system. Developed by NASA in the 1990's its function is to schedule jobs on networked Unix and Lunix systems.

Open PBS offer two versions of PBS software; an open source version and a commercially licensed 'professional' version. The professional version is promoted as having better scalability with large numbers of CPU's (more than 100's) and coping with high usage. Support for Windows and Mac operating systems, better scheduling, fault tolerance, job accounting and support for computation grids are features of the commercial version of PBS.

Probably the most important quantity that needs monitoring for a cluster is the temperatures of both the CPU and system. The unwelcome consequences of high temperatures are discussed in 5.1.2. Monitoring software should be able to acquire temperature data from the machine chassis, aggregate values from all the machines in the cluster and finally respond accordingly. In some

cases, mainly at high temperature, the response is to shut down the cluster to prevent damage through over-heating.

Currently there is no one 'killer application' for cluster management. For small clusters (40-80 machines) open-source software such as Oscar (see [49]) and Rocks (see [50]). Large clusters of machines can be trickier to run.

5.2 Principles of a Computer Network

Networks allow computers to communicate with each other. Connections between machines allow information to be exchanged between them. The concepts covered in this section are relevant to Spitfire and DUCK. The interface between MAP and DUCK is a socket-based connection protocol. Both projects were structured as web applications and have multiple tiers.

5.2.1 Packet Switching

Most modern computer networks use *packet switching.* The description of which is best given by quoting Paul Baran's US air force study (see [51] for a list of some of the author's publications);-

> "Packet switching is the breaking down of data into datagrams or packets that are labeled to indicate the origin and the destination of the information and the forwarding of these packets from one computer to another computer until the information arrives at its final destination computer. This was crucial to the realization of a computer network. If packets are lost at any given point, the message can be resent by the originator."

Packet switching networks were developed by the US Department of Defence and are designed to be decentralised and have the ability to function if part of the network was disrupted. The result was *Transport Control Protocol over*

Internet Protocol (TCP/IP). TCP/IP makes it possible to connect heterogeneous networks together to form internets. It acts as a common foundation on top of which protocols such as *Telnet, UDP* and *FTP* are built.

5.2.2 Sockets

The abstract concept of sockets is taken from the UNIX operating system and has appeared on the majority of modern operating systems. A socket is a programming interface that isolates the developer from the low level implementation of TCP/IP stacks.

Sockets are 'handles' to links that allow communication over a network to a remote application.

For TCP sockets (sockets that use the TCP/IP protocol) the following information is usually required: -

- The IP address of the remote system
- The port number on which the application is responding
- The IP address of the local system
- The port number the local application is bound to

There are two flavours of socket; *client sockets* and *server sockets.* Client sockets form connections to server sockets, which are created to listen for client connections. Once a client connects to the server socket another socket is created to handle that clients request. In that way the server socket is left free to listen for more client connections. Once connected, streams of data, for example bytes and text, can be read and written to and from the connections.

5.3 Layered Model

The TCP/IP architectures divides, broadly, into 4 layers; the *link layer*, the *network layer*, the *transport layer* and the *application layer.* Each layer has

a particular function. The *link layer* can be further expanded into the *data link layer* and *physical layer.* The *application layer* can be expanded into the *session layer* and *presentation layer.*

5.3.1 Link Layer

As stated the *link layer* can be sub-divided into 2 more layers; the *physical layer* and *data link layer.* The physical layer is concerned with sending the bit stream across the network; it provides the hardware for sending and receiving the data; this includes cables, interface cards and other related issues. Examples of such low level protocols are *ethernet*, *token ring*, ARCNET, FDDI and *Asynchronous Transfer Mode* (ATM).

The data link layer is concerned with the format of messages exchanged over the network. In this layer bits are decoded and encoded into data packets, the layer also handles physical errors that arise in the physical layer.

5.3.2 Network Layer

The *network layer* allows information to be transmitted to any machine on a connected TCP/IP network, even if the physical properties of the two networks are different. *Internet Protocol* (IP) transmits data within the layer. Switching and routing are implemented in this layer, along with mechanisms to control packet sequencing, congestion, error handling, addressing and many other functions.

5.3.3 Transport Layer

The *transport layer* provides network applications with a transparent means to transport data over the network. To do this, clearly defined channels between hosts are provided. Depending on the protocol end to end error recovery may be provided. Examples of transport layer protocols, in the TCP/IP suite, are

the Transport Control Protocol (TCP) and User Datagram Protocols (UDP) protocols.

5.3.4 Application Layer

This layer provides applications with the means to communicate. All functionality in this layer is application orientated. Architectures for tiered network applications (see 5.4) are part of this layer. The layer can have session and presentation (Syntax) layers introduced as the lower layers.

Examples of protocols in this layer are HTTP (Hyper Text Transfer Protocol), FTP (File Transfer Protocol), and Telnet.

5.4 Tiered Models

Web applications and network applications tend to be split into tiers. A tiered architecture means that the functionality of the application is spread across multiple tiers, which could be running on separate machines. The most well-known is the client server model.

There are other models of network computing, examples being; -

- Mainframe. Centralised computing. All the computing power resides on the mainframe. Clients are dumb terminals that display output and capture input only.

- Peer to peer (file sharing). No machine has overall control and machines can initial requests or service requests.

5.4.1 Client-Server Model

The *client-server model* is the most basic form of network application model. It involves having a server that serves a number of clients. The client-server model is a two-tiered architecture. During typical operation the client program

makes a request to the server program, which the server program responses by servicing the clients request. Although a client server program pair can be set up on a single computer, client server applications usually run across networks.

Server programs sit and listen for clients to connect. Software components often referred to as daemons are used to listen constantly for connections.

5.4.2 3 Tier Model

The 3 tier model was devised to overcome several limitations of the 2 tier model. In a two tier system heavy processing on the client may generate large amounts of network traffic. Futhermore, the two tier solution has problems when it comes to maintenance; even small changes require a re-deployment right across the user base.

The advantage of the 3 tier model is that it uses well-defined interfaces to break the application up into a 3 tier/layer architecture. The tiered structure comprises of; -

- Presentation logic tier. Often a Graphical User Interface
- Application logic tier. Also known as the business logic.
- Data logic Tier. Contains the data source for the application which is often a database.

5.4.3 Multi-tiered Model

Although the three tiered architecture solves many of the maintenance, performance and network traffic problems. It results in 'stove pipe' solutions; meaning that the different solutions do not communicate with each other.

The n-tier architecture aims to solve this problem. By defining multiple application objects instead of the middle tier, we arrive at an n-tier architecture.

The application objects communicate with each other by using interfaces. With the middle tier defined in this way, multiple applications can be built up re-using the components in the middle tier.

5.4.4 Web Application Model

The architecture of a web application tends to follow the 3-tier architecture, the tiers being as follows: -

- *Presentation Layer*, which includes the browser and web server. This is because the web server is responsible for turning the data into a presentable format.
- *Application Layer.* A program (or script) that provides the functionality of the application.
- *Data Layer.* This layer provides the second tier with the data it needs.

5.5 Content and Service Delivery Mechanisms

5.5.1 Web Servers

A web server, in the hardware definition, is a machine that handles web requests. In software terms they are daemon programs (see section 7.4.3)

These requests are HTTP requests, which request for web pages (or indeed many other documents as defined in the MIME standard media types [52]). A client (user using a web browser) is responsible for generating the request. The response to the request is to return the document that has been asked for.

Clients make their requests in the form of a *Uniform Resource Locator* (URL), which is better known as a web address; for example, `http://www.cern.ch/`.

The most popular web server software is the *Apache Httpd package*, [53].

5.5.2 CGI

The Common Gateway Interface (CGI) addresses the issue of providing interactive or dynamic content. It provides a standard for allow web servers to pass requests to executable programs running on the server. In this way requests can be serviced with program executions and output and not just static HTML pages.

The program executed in the process of a CGI call is run on the server in a similar way to any other program. The CGI acts as a gateway to the programs you wish to make available. CGI can take the form of programs, which need compiling first and scripts which just need to reside in the CGI directory. The programs/scripts can be written in any language that will run on the system, Perl is a popular choice.

When a web server receives a request that requires a CGI call a number of processes are started. Firstly, the web server creates a process in which the CGI program/script can be run. Any runtime environment and the program/script itself can then be started. The request is passed to the program/script in the form of an environment variables such as `QUERY_STRING` and `PATH_INFO`, as opposed to command line argument. The program/script is then run with these parameters and the response from stdout is returned to the user.

A big problem with CGI is scalability. Due to the fact a new process is created for each request a large number of requests could easily swamp a system. Solutions have been created to shared single instances of programs, which eases the problem. Programs running on the server give rise to a number of security issues. To resolve these issues, precautions such as running CGI programs/scripts in a designated folder and not displaying them to the user are taken.

5.5.3 Servlets

Servlets are Java programs. Java provides a number of ways to deploy code, the reader may be familiar with some or all of them see section 3.3.6. Java servlets are just one of many ways to deploy Java code. Servlets provide advantages over traditional means of providing interactive and dynamic content on the web; -

- efficiency, the threading model allows concurrent requests to handled efficiently.
- Platform independent (like Java).
- have access to the Java API, providing a rich API of database access, JavaBeans, XML, CORBA, etc.
- Servlets don't just have to return HTML. They can return a whole number of formats including images, zip files, audio clips and binary files.

Servlets are made available to the outside world by deploying them in a servlet container or servlet engine. The servlet container is responsible for initialising the servlets, communicating requests and responses to the client and providing a framework of infrastructure for the servlets. To that ends a servlet container is built around a *Java Virtual Machine* (JVM) and possesses the ability to handle network requests from clients.

Typically services such as database connectivity and XML processing are available to servlets in a servlet container. Servlet containers can support Java Server Pages (JSP) as well as servlets. JSPs are web pages with nuggets of Java code embedded. The pages are processed and compiled the first time they are requested. Tag Libraries can be used in conjunction with JSPs to help their reuse.

The servlet API is a package of classes and interfaces for producing servlets. The package provides interfaces and abstract implementations of servlets so that the servlet programming is not concerned with such tasks as creating

network sockets etc. Amongst other things, the API provides two abstract implementations of the interface servlet; one is a generic servlet which allows servlets that communicate with various protocols to be written and the other is the `HttpServlet`, which allows more HTTP-specific servlets to be written. Indeed the whole API is divided between `javax.servlet` with classes and interfaces for more generic servlets and a `javax.servlet.http` for HTTP servlets.

A servlet has a life cycle, which is dependent on the servlet container and defined by the `javax.Servlet.Servlet` interface. The servlet container is ultimately responsible for the creation and destruction of servlets.

The life cycle of a servlet is typically as follows;-

- The servlet container initialises.
- If the web application configuration says the servlet should be loaded on startup, an instance of the servlet is created.
- Otherwise, the servlet engine waits until the first request before an instance of the servlet is created.
- The *init()* method is called on the newly created instance. Here the servlet can perform any tasks it needs to do once to prepare itself for handling requests. The tasks may include loading configuration information, creating database connections etc.
- When a request is made to the servlet, the *service()* method is called. This allows the servlet to service the request. The *init()* method is guaranteed to have been called before any *service()* requests are made.
- Before destroying an instance of a servlet, the *destroy()* method is called.
- The servlet is then marked for garbage collection

In theory the life cycle could be performed for each request, although this would be very inefficient. A better approach is to create an instance of a

servlet and reuse it to handle requests. Servlets tend to be destroyed when the servlet container is shutdown or the servlet has not been used for a while.

In the event of the *service()* method being called whilst the servicing of the request is already in progress, the servlet container will create another thread of execution. This means servlet code has to be what is known as thread safe.

In practice servlets do not create a new thread for each new concurrent request. Instead a pool of servlet instances is created and requests allocated an instance from the pool.

Servlets do not tend to be deployed on their own. Most applications are built from a number of servlets, each servlet performing a particular task. Web applications are groupings of servlets.

A web application consists of;-

- a root folder.
- the servlet classes and other non-servlet classes under the folder `/WEB-INF/classes`.
- a file `web.xml` under the folder `/WEB-INF/`. This file describes the servlets in the web application and their mapping to URLs.
- Other content, such as static web pages and images.

Web applications can be deployed as a collection of folders or as special `.WAR` files. `WAR` files (or Web ARchives) are similar to jar and zip files and contain the web application. The servlet container unpacks the war file when it deploys the web application.

5.5.4 Web Services

Introduction

Often it is required that the concepts of component based computing are scaled up to the network. On intranets and internets applications often access software on remote machines. This technique is known as a *Remote Procedure Call*

(RPC), see section I. Being a popular method of implementing client-server solutions, there are many implementations of RPC. The RPC model is complicated because of the possibility of heterogeneous models of computing; the RPC calls may be a computer of different hardware or even running a different operating system. Web Services are to address many of the problems of the existing RPC methods. They do this by, amongst other things, using existing technologies (XML for the encoding scheme, as visited earlier), decoupling from programming languages, transport protocols and environments and most importantly providing an open standard.

Motivation

The short-comings and lack of adoption of existing protocols provided the motivation for web services. There have been a number of different RPC style technologies, all possessing some failings to a certain degree. Some RPC style technologies end up being tied to one particular platform or language, limiting their use and up take. Other protocols are inherently connection based and can not handle network disruption. Another major problem with the transport protocols is connecting through networks. Often RPC protocols use non-standard ports which tend to get blocked by firewalls in the increasing security conscious corporate intranet environments. Such problems have often lead to software vendors building their own infrastructure to support RPC. This is obviously an undesirable situation.

We now have the motivation for a set of standards that try and solve the problems above in a widely accepted fashion.

Architecture

The web services standards are considered to be composed from five technologies;-

- Discovery Mechanisms

- Description Mechanisms
- Messaging structure
- Encoding
- Transport Protocols.

The first three technologies provide the core architecture for web services. They are XML based and hence, not tied to any one programming language.

A client needs to discover what services, offered by providers, are available to it. *Universal Description and Discovery Integration* (UDDI) provides a central mechanism for clients to match their requirements with web services. Developed by a consortium of large technology industry companies; UDDI uses a centralised, hierarchical directory mode to implement its services. *DISCOvery of Web Services* (DISCO), in contrast provides a less centralised model for the discovery of web services. Both UDDI and DISCO use XML.

Web Services Discovery Architecture (WSDA)[54], provides an architecture to perform the same functions in a grid environment.

Using discovery mechanisms an end point for a web services can be resolved. The client then needs to work out how to interact with the web service. Put basically once a web service is found, the client then needs to work out how to use it. *Web Services Description Language* (WSDL) provides a means to do this. Based on XML, WSDL documents are a layer on top of the schema describing a web service. The provided web service is effectively documented by the WSDL document. The syntax of a WSDL document is as follows;-

- the *definitions* element is the root element of the document and uses the WSDL namespace, allowing entities to be defined.
- *types* elements allows the types of the data used in the web service. The default system to define types is XML schema.
- the *message* elements associate the message with its definition in the schema.

- *portType* describes which interfaces in the software the web service can expose

- *binding* links the portTypes to a particular protocol, for example the SOAP protocol.

- *service* defines the endpoints exposed by the web service, for a web service using the SOAP protocol this would be a SOAP address.

Simple Object Access Protocol (SOAP) is fairly central to web services; it is one of many RPC style communication protocols. This begs the question why has another protocol been concocted for web services? There are a number of reasons;-

- SOAP is transport protocol agnostic, meaning it does not require any one particular transport protocol to provide the mechanism to transfer SOAP messages. Therefore SOAP can use any number of common transport protocols. A popular choice is HTTP, but STMP and many others may be used.

- SOAP has no tight coupling to any programming languages. This is because the SOAP specification does not define an API. The API has to be specified by the programming language or platform making and receiving the SOAP messages. Being XML part of the functionality required to specify and implement the APIs required is provided by modern programming languages. Indeed, there are a number of SOAP implementations for the popular modern programming languages.

- SOAP is not trapped on any one particular platform. SOAP-based web services can be deployed on Linux, Windows, UNIX, etc. boxes.

Web services use XML for encoding, which is character based as opposed to being a binary format. XML provides a cross-platform format which supports most (if not all) character sets. By not using a binary encoding schema binary

issues such as big and little endian data are avoided[1]. Using XML means data is processed by the wide range of parsers available on many platforms.

SOAP describes, in its specification, how SOAP messages should be transported over HTTP. However, SOAP is not limited to HTTP as a single transport protocol; any way an XML document can be transmitted over the network is available as a possible transport protocol. Another popular choice for a transport protocol is *Simple Mail Transfer Protocol* (SMTP).

WSRF

Web Services Resource Framework (WSRF) can be thought of as the convergence of web services with grid computing. WSRF improves on web services by introducing the standardised notation of stateful-ness; a concept not present in web services. Web services often allow users to access resources which have a state; meaning between requests to the resource values, variables and data structures, persist. However the concept of state is not explicit in the definition of web services. WSRF is designed to address this issue. A WS-Resource is defined as a stateful resource accessed by a web service. So in WSRF each service has a resource object to stored variables. The stateful resource should be:-

- expressed in an XML document which gives the type which will have a web services portType associated with it.
- The resource should be accessed in a fashion consistent for the resource pattern. This means using WS-Addressing for the end point references.

The WSRF framework is divided into 5 specifications, which are listed below;-

- - WS-ResourceLifetime
- - WS-ResourceProperties

[1]Big endian and little endian refer to the different ways numbers are stored in the machines

- - WS-RenewableReferences
- - WS-ServiceGroup
- - WS-BaseFaults.

Globus (see [14]) provides a Java and C implementation of WSRF in its Globus Toolkit, the aim being to express grid services in terms of WSRF. This makes it easier to use mainstream technologies such as WSRF in grid and also makes grid more attractive to mainstream computing.

Hence we have come full circle and arrived at grid from more mainstream technologies as opposed to arriving from an almost theory based direction.

5.6 Summary

Has the grid superseded the cluster? No, clusters are still valuable resources that complement the grid. They offer a concentrated pool of processing power that can be seen as a 'grid resource'.

Clusters tend to be localised in one room, as opposed to being distributed, and have homogeneous hardware and software. They also tend to be administered by a single administration and have a (normally) fixed number of processors, such as disk pools. This means that the middleware for running a cluster is simplified compared to the middleware for running a grid.

Some schools of thought might classify clusters as a special case of grid. The two should perhaps be seen as complimentary resources for providing computing power.

The concepts of computer networking and structuring of applications have been introduced in this chapter. Socket based communications and the underlying infrastructure provide a means for communication between computer systems. The tiering and structuring of distributed software allows it to be more easily developed and deployed and also helps the scalability of the finished solution.

Chapter 6

A Web based browser for Spitfire

6.1 Introduction

This chapter discusses the writing of a web-browser based element for the *Spitfire project*; part of the *EU-Datagrid* project[55]. The EU-Datagrid is a European Union funded project aimed at producing a grid (see chapter 2) and hence, an E-Science infrastructure for the European science community. The EU-Datagrid project was brought to completion in March 2004, with *Enabling Grids for E-Science* (EGEE)[7] continuing where it left off.

The EU-Datagrid is divided into 12 work package groups;-

- WP1 Work Scheduling
- WP2 Data Management
- WP3 Monitoring Services
- WP4 Fabric Management
- WP5 Storage Management
- WP6 Integration Testbed and support
- WP7 Network
- WP8 Particle Physics
- WP9 Earth Observation
- WP10 Biology
- WP11 Dissemination
- WP12 Project Management.

Work Package 2(WP2) is associated with *data management* on the grid. This involves information resources such as large databases. The aim of WP2 is to provide grid-enabled access to large amounts of structured information. This work package handles such issues as ease of use and transparency, along with the heterogeneous nature of resources and the scalability of datasets of order of peta-bytes.

The WP2 group has tackled these challenges by studying the above issues and producing and deploying software solutions, typically in the form of *grid middleware*(see 2.4). Areas tackled by the WP2 group include; replication, metadata storage, security and optimisation. The software solutions produced are, amongst others, the *EDG Replica Manager*, the *RLS Replica Location Service*, the *Replica Metadata Catalog*, the *Replica Optimisation Service*, the *OptorSim* and the *Spitfire Client and Server software.*

Spitfire needs to access relational databases in a scalable, secure, transparent way that allows for heterogeneous resources.

The Spitfire middleware consists of client and server modules. On the server side it takes advantage of the existing relational database connectivity provided by JDBC (see 4.7.4). This means the Spitfire server component can, in theory, talk to any JDBC enabled relational database.

The connection protocol from client to the server is provided by SOAP (see appendix I.6). The functionality provided by Spitfire is a set of definable database operations. Which operations are available to a user depend on their privileges.

The implementation of the Spitfire project uses existing software components where possible and writes new components to provide the 'glue' to bind everything together. Existing packages used in Spitfire include Apache AXIS for the SOAP Implementation (see appendix I.6) and MySQL relational database as a database back-end.

Java and C/C++ client APIs are provided by Spitfire to use the services it provides. This means that clients can be programmed by developers in the

Java and C/C++ languages. Security is transparent, meaning the developer only needs to worry about importing the correct packages or linking the correct libraries and then using them. Typical Spitfire queries only involve a few lines of code.

The Spitfire API for Java is split into 3 interfaces, `SpitfireCore` which provides the core interface for Spitfire, `SpitfireInfo` which is a database information service interface and `SpitfireAdmin` which is a database administration interface. Applications are able to use Spitfire by writing code that calls the methods in these interfaces.

There are two main alternatives to Spitfire that I will discuss, these are SRB and OGSA-DAI.

The *Storage Resource Broker* (SRB)[56] has been developed by San Diego Supercomputing Centre and is often used in conjunction with *Metadata CATalog* (MCAT). It provides middleware for accessing heterogeneous data sources over networks. SRB has a wider scope than that of Spitfire, which is designed to handle only relational databases. The storage systems SRB can access include HPSS, UniTree, ADSM, Unix File System, NT files system, Mac OS X File System and databases. SRB also supports a wide range of APIs for programming languages including C, C++, Java and Python.

The Open Grid Services Architecture and Data Access Initiative (OGSA-DAI)[57] is an open-source collaboration between industry and academia. It allows access to structured data in both *relational databases* and *XML* data sources. The middleware is based on the Globus Toolkit and is very much grid services orientated.

OGSA-DAI reuses existing standards and technologies to provide a *grid data service*. These standards include OGSA, Java along with existing transport and query protocols. The abstractions behind the grid data service allow different data sources to be used, such as Xindice an XML database by the Apache Project[45], MySQL relational database[58], Oracle relational database and DB2 relational database.

Over the two solutions discussed above, Spitfire has advantage is being relatively lightweight; there are a minimal number of components that need installing to 'gridify' a relational database. It does, however, have the downside of only supporting relational database; however, this may not be a problem for some applications.

6.2 Spitfire Web

The result of my project to produce a proof of concept web (http/html, see 4.6.3) based interface for administration functions in Spitfire is *Spitfire Web.* It is middleware allowing a web browser to be used as the client. The Spitfire API is implemented in Java; therefore the most appropriate form for a solution involves the Java programming language. Furthermore Java can be used to produce web-enabled output in the form of servlets and JSP. My solution initially involves servlets and the planning for the possible use of JSP later.

JDBC is the initial interface directly to the database, cutting out the SOAP layer. Later, with the use of SOAP, the Spitfire API will be taken advantage of. In its final form the web application can act as a tier in the Spitfire system.

In terms of functionality, this web application should provide 'drill down' views of the database host, databases, tables, and examples of data contained in the table. For correctly authorised users manipulation of the database structure is possible, for example, the ability to create and drop columns, tables and databases. Users should also be able to browse the database host using a series of web page requests. This means the web application has a requirement for persistency between these requests to keep track of information like which part of the database is being browsed, which user is browsing and so forth.

In the finished system a servlet (see 5.5.3) is used to provide the user with the ability to browse the system using a web browser. JDBC (see 4.7.4) is used to provide direct connectivity to a relational database for testing. An alternative implementation using SOAP (see I.6) has been started. Details of both connectivity solutions can be seen in the following section.

6.3 Implementation

In this section a number of specific problems that were encountered and their solutions are described. The browser is written as a single Java servlet (see 5.5.3) and packaged as a web application for ease of deployment. Aside from the servlet, the remainder of the source code for the solution is split into 9 classes and 1 interface.

The single interface `SpitfireConnectionLayerBasic` defines the functionality of the servlet. Beneath this interface two implementations are planned, the test JDBC implementation and the SOAP implementation.

HTTP is a stateless protocol, meaning no information is retained between requests from the client. The database browser has to maintain a certain amount of information between page requests. This information includes the user that is browsing, the host they are logged into, the database in the host they are using and the table in the database.

In addition to this information, we need to be able to send information requesting certain actions be carried out. These actions might be requests to navigate to views (databases, tables, details of tables etc) or requests to drop/creating databases and tables.

There are a number of well-known ways that provide state or persistency for HTTP which are *cookies*, *server side session management*, *hidden forms* and *URL rewriting*.

The chosen solution is to encode the parameters into URLs; this is known as URL re-writing. Each link in the web page has a set of parameters encoded in the URL, these are appended after a '?' symbol at the end of the URL. The parameters are presented to the server when the user clicks the link. This means the page sent to the client has to have all its URLs appended with parameters for requesting actions. There are a number of classes to handle the encoding and decoding of these parameters as they were exchanged with the client.

Figure 6.1: The web page which allows users to log into Spitfire Web and specify the database to use.

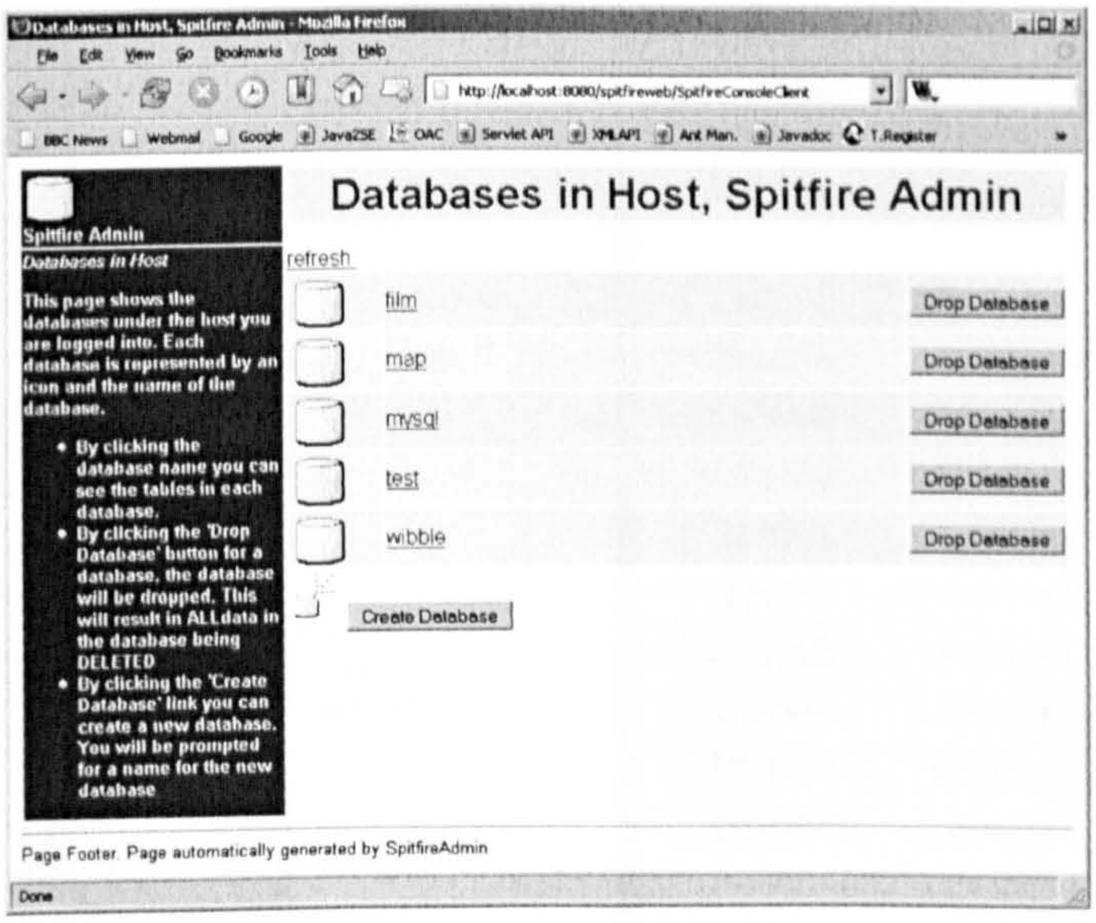

Figure 6.2: A web page showing the databases in the host the user is connected to.

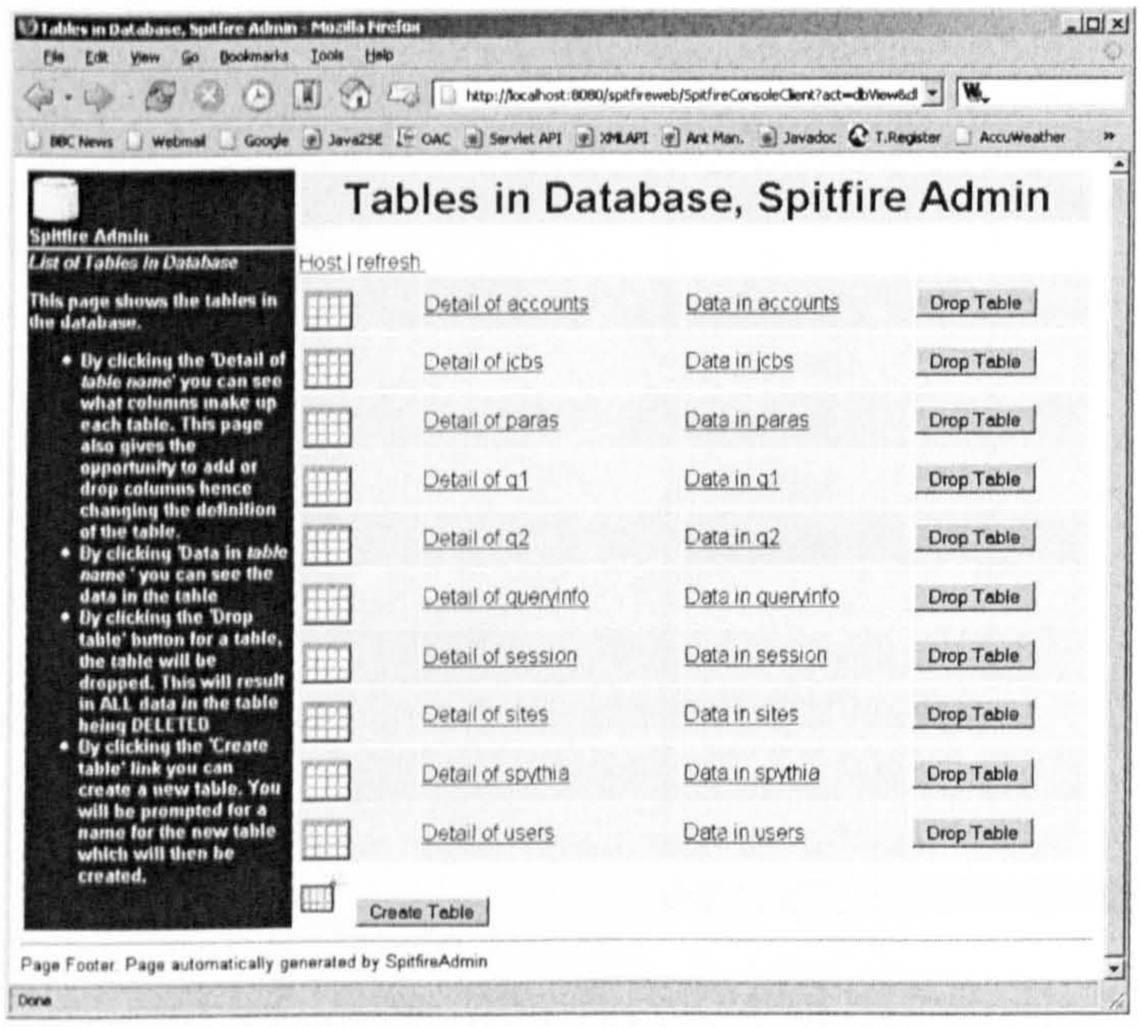

Figure 6.3: Web page showing the tables in the selected database.

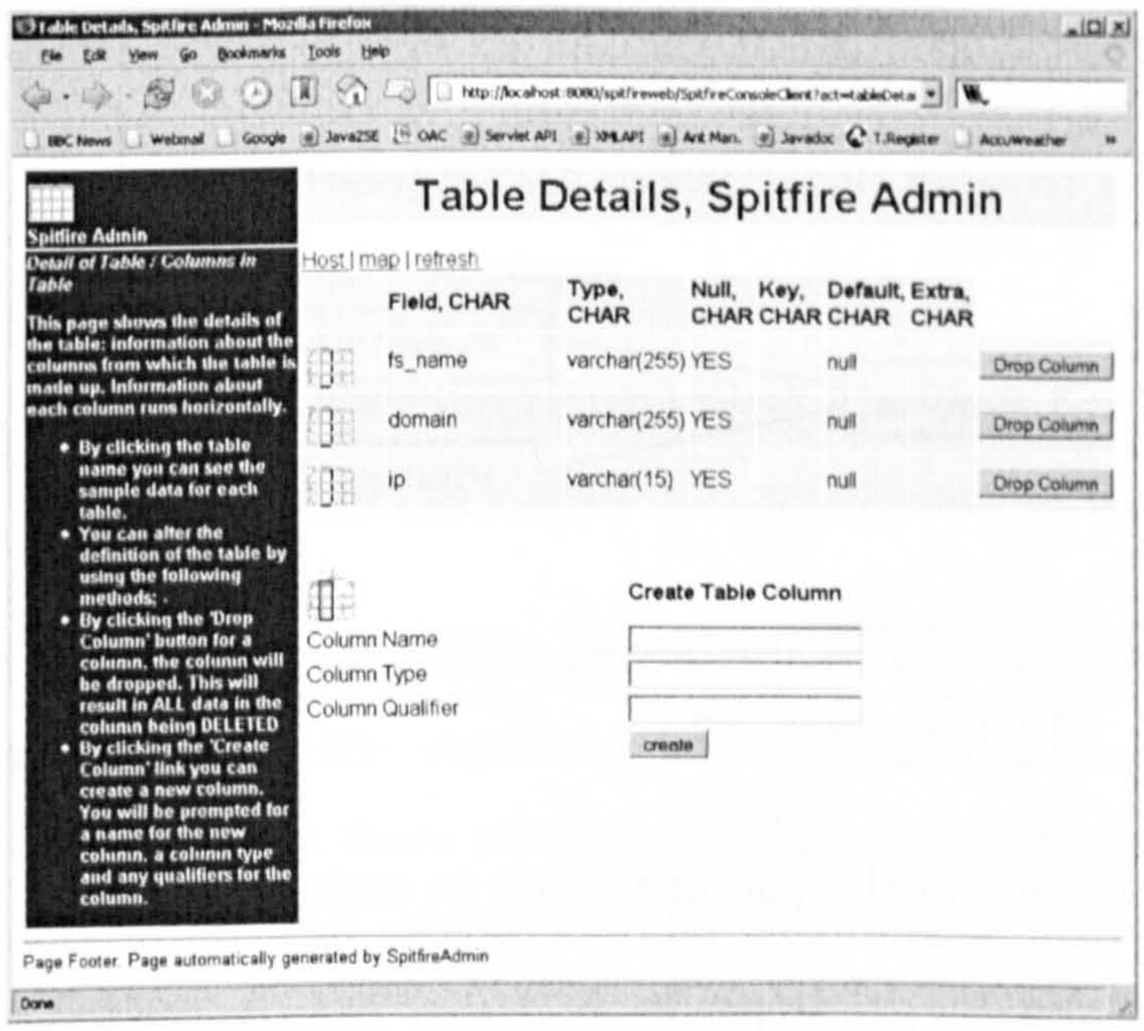

Figure 6.4: A web page showing details of a selected Table.

6.4 Classes and Interfaces

This section describes how the classes and interfaces build up the web application. The components of the web application are housed in a single package which has the 'namespace' `org.edg.data.spitfire.WebBrowser`. This fits into the naming scheme adopted by the Spitfire project and indeed the rest of the EDG project.

Figure 6.5 illustrates the way in which classes within the package relate to each other. The `SpitfireConsoleClient` is a servlet that is the user interface to the system. `SpitfireConnectionLayerBasic` is an interface that `SpitfireConsoleClient` uses. This interface is to make the package pluggable between implementations that act as the business logic. `HTMLLogic` uses `SimpleHTMLPage` and `HTMLTable` to create the HTML output that `SpitfireConsoleClient` outputs. `URLStringDecoder` and `URLStringEncoder` are for reading and writing data into the URL, acting as a persistency mechanism.

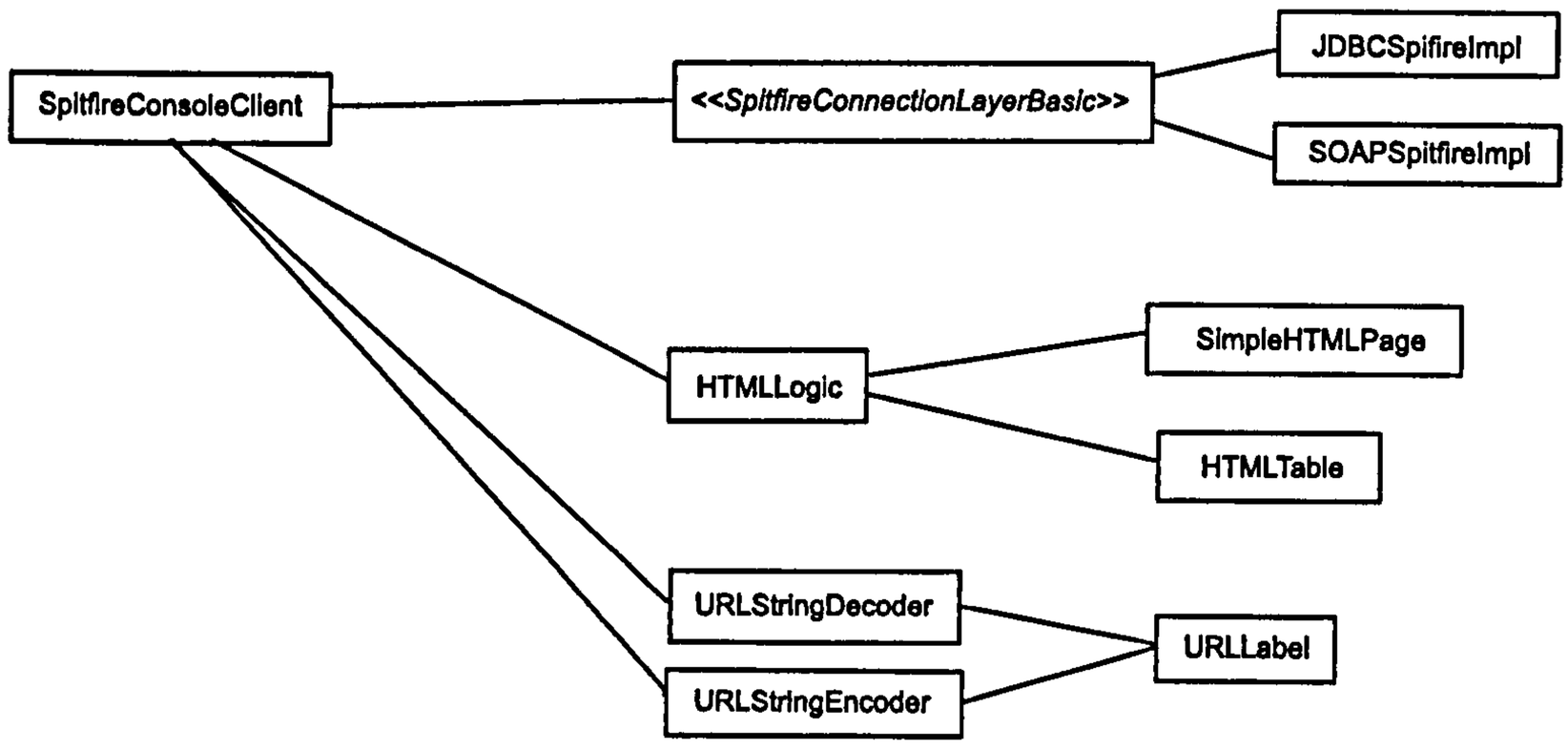

Figure 6.5: How the classes in the package relate to each other.

6.4.1 HTMLLogic

The `HTMLLogic` class provides a wide range of functionality. It encapsulates the logic for producing HTML pages; the user interface for the web application. There are methods for outputting web pages for various situations and presenting different types of data. Examples include, presenting database table data to the user, getting the user's input and providing feedback. Various snippets of Javascript that provide functionality for the web pages are also generated by the class.

There are a number of items need that need to be passed to the constructor to create an instance of `HTMLLogic`. These items are stored using the keyword 'final' inside the class, rendering them *immutable.* This means they cannot be replaced with new instances for the lifetime of the `HTMLLogic` instance, therefore the values passed to the constructor are retained for the lifetime of the instance of the class. This behaviour was considered appropriate for the instances.

The items (instances of these classes) are

- URLQueryStringMaker- an instance of the class which makes query strings, see the section 6.4.7.

- String (servletPathToSet) - the path of the servlet as a URL.
- String (imagesPathToSet) - the path of the images to include in the outputted web pages.
- String (styleSheetPathToSet) The path to the CSS-compliant stylesheet that is used to specify the appearance of the outputted web pages.

The constructor also makes a call to the *System.getProperty()* method that retrieves the line separator for the platform the web application is running on. This is a more portable method than simply hard-coding *newline* characters into the program. The *getProperty()* method uses the property key `line.separator` to return the platform-specific line separator.

The *doPage()* method is private, meaning it can only be called by other methods within the class. It produces the HTML source for a web page using the arguments supplied to it, resulting in web pages have a standardised look and feel. The method returns the HTML source for a web page encapsulated in a string.

The outputted page consists of various elements, see the figures 6.1, 6.2, 6.3 and 6.4. One of which is a corner icon; a small image placed top left of the page, which can represent the database or the database table. Below this icon, on the left 25% of the page is a pane, which contains contextual information and help. In the remaining 75% width of the page to the right is a grey title pane, a navigational bar and the actual content of the page. A page footer runs across the bottom of the page. The title bar shows the same title as the window title. Links to other pages are given in the navigational pane which changes depending on the web page.

The method requires the following parameters;-

- The page title as a string
- The HTML for the navigation line (as a string)
- The path inside the web application file system to the corner icon

- The HTML for the main contents of the page (as a string)
- The HTML for the help pane (as a string)
- Any Javascript that needs including (as a string).

The method *makeTablesDetailsPage()* produces a detailed view of the requested database table as a web page (see Figure 6.4). Embedded in the page are links to navigate back up to the enclosing database and host. The method *doPage()* helps put together the whole page in a standard format consistent with other web pages produced by the web application. The resulting page also has options to drop and create columns. Column details needed to create a column in a database table are; the *name* of the column, the column *type* and the column *qualifier*.

The method *makeLoginPage()* creates a login page, allowing the user to log into the web application (see figure 6.1). The method uses *doPage()* to produce the page in a standardised fashion. The output is a web page that contains a form in which the user can enter their login details; the *host*, the *user* that is logging in and their *password*. The form to enter the login details is displayed on the majority of the page. Down one side of this is a pane displaying instructions for logging in. This information is loaded from a static file using the method *readPage()*.

These details are then passed back to the web application using the POST method; which transmits data as plain text over the network. This method of transmitting the details is not secure and would need changing for a production system.

The method *makeTablesViewPage()* creates a page showing every table in a given database (see Figure 6.3). It is also responsible for creating the links from the page so the user can navigate to the tables contained in the database. The page presents options to perform various actions on the database and the tables located in it. The tables in a given database are listed on a table per row basis. Each row has two links associated with it; one to show the detail of the table and another to show a sample of the data in that table.

The outputted web page has a 'Drop Table' button on each row together with the description of the table. The detail required to drop a database table is simply the table name. A Javascript dialog will ask the user to confirm that they wish to delete a given table. The confirmation that the action has been performed is displayed in a new window, which the user can close.

A new database table can be created by clicking the 'Create Table' button located at the bottom of the web page. A Javascript dialog prompts for a name for the new table. Once the table has been created a confirmation message appears in a new window.

Above the table displaying the list of database tables is a navigation bar with links to go back to the host (the list of databases in the host) and a link to refresh the current data. Along the side of the web page is a panel giving the user instructions and information, which is loaded from a static file using the method *readPage()*.

The method *makeDatabasesViewPage()* creates a page that shows all the databases in the host (see figure 6.2). A database is listed on each row, with a link to navigate to it and a button ('Drop Database') to delete the database. If the user decides to click the button to remove a database, a Javascript dialog prompts for confirmation before the database is removed. There is also a button to create a database, this is situated at the bottom of the page. Clicking on this button produces a Javascript prompt asking for a name for the new database, with confirmation of its creation given in a new browser window. Above the table row listing the databases in the host there is a navigational link to refresh the page. The page may need to be refreshed after databases have been created or deleted. Along the left side of the page is a pane providing help and instructions for the user. This information is loaded from a static file using the method *readPage()*.

The final html is produced using the method *doPage()*.

A number of methods were produced to generate Javascript functions for placement in the web page source code. Whilst having the Javascript in the source code is a little inelegant, parts of the generated Javascript are needed to be

changed at runtime. The *makeDropJavaScript()* method produces Javascript for dropping databases, tables and columns.

The method *makeNewColJavaScript()* makes the Javascript for the creation of new columns.

The method *makeCreateJavaScript()* creates a generic Javascript from prompting for names for new items, for example tables, databases and columns. The user sees a dialog box allowing them to give the new item a name.

The method *makeDataSamplePage()* provides the user with a sample of the data in the database table. The method accepts the sample data and the header, a set of column titles, in the form of an array of strings, as arguments. Data is formatted into HTML table which displays the cells of sample data. The method produces a header showing the titles of the various columns, and outputs the rows in alternating colours to ease readability of the data. *doPage()* is used to compile the final output; which is HTML encapsulated as a string.

makeErrorPage() is a method for displaying a message warning of an error situation occurring. Like the confirm page method this is a 'one liner' method wrapping a call to the *doPage()* method. This method accepts a stack trace object as one of its arguments and prints it as part of the web page returned to the user.

An improvement would be an email contact link, so users could provide feedback or notify the web applications administrator about the error situation.

The *makeConfirmPage()* method is used to produce a confirmation page for certain actions. This page confirms that the actions have taken place and/or the user's request has been carried out. To produce the final output as a web page the method *doPage()* is used. A null value is passed to *doPage()* for the Javascript, as Javascript is not required in the confirmation page. Usually the confirmation window opens in a new browser window, which can be closed by the user. This behaviour is due to commands in the Javascript elements being embedded into the page.

The private method *readPage()* is used to read in existing web pages. A static web page that exists inside the web application can be read. An `inputstream` is provided using the method *getResourceAsStream()*, as provided by `java.lang.Class`). After checking that the `InputStream` is non-null, an `InputStreamReader` (which is buffered by a `BufferedReader`), is used to read the input stream. Using the method *readLine()* in the `BufferedReader` class, the page is read a line at a time and each line is appended into a StringBuffer.

When the *readLine()* method returns *'null'* the end of the file has been reached. Building up the file in a StringBuffer is more efficient than simply concatenating strings as each line is read. This is widely regarded as inefficient because strings are immutable objects and concatenating them involves the creation and destruction of many objects.

The *readPage()* method returns the file in a string which is created by calling *toString()* on the `StringBuffer` when the reading of the file is complete. Any errors are handled in this method by returning a message of explanation in the string. The method is mostly used to read the supplementary panes of information for web pages in the web application.

6.4.2 HTMLTable

The class `HTMLTable` has three constructors and encapsulates logic for serialising data to HTML tables. Displayed data is not passed to the constructor; instead the constructor is more concerned with attributes that affect how the data is displayed. The method *toHTMLString()* produces the HTML table and returns it ready for use in a web page. The method *appendCellHTML()* is used to add each cell to the table. When a row has enough cells (columns) the *closeRow()* method is called to finish the row and start a new row.

As stated previously HTMLTable has 3 constructors. The first of these accepts just one string as a parameter, this string should contain the attributes for the table. The second constructor accepts the same table attributes and also accepts an array of strings as an argument. The array contains different sets of

attributes for different columns. For each column in the table the class applies a different set of attributes from the array. As tables are rendered in a cell by cell fashion into rows and then rows into the completed table, the attributes are alternated as the code for each cell is generated. The third constructor allows attributes for the table to be set and attributes to be set on both a column-wise and row-wise basis.

The method *applyAttributesToEachCell()* allows the attributes for each table cell to be set as a string.

The *firstRowAttributes()* method sets the attributes for the first row of cells in a table, the attributes are then not used again in the table. The method is useful for creating a title-like row, or if the HTML table header has been used, a secondary description row.

Once the HTMLTable object has been constructed, HTML for each cell can be appended using the *appendCellHTML()* method. This method accepts the HTML as a string, which is then stored internally in a vector. This vector is used to store a row of cells. When the method *closeRow()* is called the vector is processed, the cells turned into HTML and added to the table body HTML. An integer is returned for each cell added using *appendCellHTML()* method, this indicates the position of the cell in the row.

Once a row of cells has been added to the table, the row can be finalised by calling the *closeRow()* method. After this method is called the next row can be started. Internally the method constructs the HTML for the row of cells that have just been added, it then appends the HTML to an internal StringBuffer.

As a row is being constructed the cells are added to the row vector. When the *closeRow()* method is called a sequence of processes occur. The vector containing the cells is checked to ensures it is non-null and of non-zero size. The HTML for the row is then generated, along with any attributes. The row vector is read, element by element and the contents turned into table cells. Each table cell may have attributes. The resulting HTML is appended to the table HTML and the contents of the vector cleared so that the vector is empty for the next row of cells. A variable that counts the number of rows is then

incremented.

The method *makeAttributes()* creates a StringBuffer for the attributes.

The method *makeTableCellHTML()* is called from within the *closeRow()* method to construct the table cells that have been added by calls to *appendCell-HTML()*. In the *makeTableCellHTML()* method the attributes for the cell are worked out and added. The method returns the HTML for the table cell in the form of a string.

`makeTableOpenHTML()` makes the HTML for the opening of a table. This includes the opening table tag which may include attributes relating to the entire table.

The *makeTableCloseHTML()* method makes the closing HTML for the table. It is a lot simpler than *makeTableCloseHTML()*.

The *toHTMLString()* method constructs the HTML for the entire table. It builds the HTML up into a StringBuffer, which is more efficient than string concatenation, as discussed earlier. The process of building up the HTML starts with opening HTML after which the table body is appended and finally the closing HTML is added. The resulting HTML is returned encapsulated in a string.

6.4.3 SpitfireConnectionLayerBasic

The `SpitfireConnectionLayerBasic` interface and its methods, define the functionality of the web application. The use of an interface means that different implementations of that interface can be produced. Hence, the same interface can be satisfied by using different implementations. For a class to implement an interface it must implement all methods that are defined in the interface.

In the case of this interface, SOAP and JDBC implementations are provided. The interface also acts to provide a clear separation between the presentation and 'business' logic of the web application. The database host is assumed

to be set in the constructor; with no reference to the host in the method calls and hence, the interface. The naming of the host is considered to be implementation dependent.

The methods in the interface are declared to *throw exceptions*; when an error condition occurs the implementation will inform the calling code using the exception.

The methods defined in the interface;-

- *public String[] getDBs() throws Exception;*
- *public String[] getTables(String database) throws Exception;*
- *public String[][] getTableTasteData(String database, String* table) throws Exception;
- *public String[] getTableTasteHeader(String database, String table) throws Exception;*
- *public String[][] getTableDescription(String database, String table) throws Exception;*
- *public String[] getDetailHeader(String database, String table) throws Exception;*
- *public String createDatabase(String databaseToCreate) throws Exception;*
- *public String createTable(String database, String tableToCreate) throws Exception;*
- *public String dropDatabase(String databaseToDrop) throws Exception;*
- *public String dropTable(String database, String tableToDrop) throws Exception;*
- *public String createColumn(String database, String table, String column, String type, String qualfiers) throws Exception;*
- *public String dropColumn(String database, String table, String column) throws Exception;*

6.4.4 SpitfireConsoleClient

The `SpitfireConsoleClient` class is a single servlet that acts as a user interface. It responds to requests from the client and supplies the requested information about the database host. The client is typically a web browser, which communicates using the HTTP protocol, is stateless and supports the use of URL rewriting. The single servlet performs a number of roles depending on the nature of the clients request. The requests can be divided into two types. Firstly, ones with no details; where the user has simply called the servlet without any URL parameters. The servlet then responds by producing a web page with a form, which the user can login with. Secondly, requests by a user who is logged in i.e. a 'normal' request. The URL parameters are available, so pages are returned, which allow the user to browse the database host. This type of request is handled by the method *handleNormalRequest()*.

The servlet loads, on initialisation, the parameter for the database drivers name. This parameter is stored in the `web.xml` file and can be set by editing this file. The parameter is a fully qualified Java class name, which means the package it is in is specified.

The servlet is initialised when it is ready to be put into service. This is part of the servlets life cycle. Code in this class currently has a hard-coded switch for switching on the SOAP part of the software, this will be changed to a parameter in the future.

The *init()* method is called when a servlet container initialises a servlet. In the `SpitfireConsoleClient` servlet the *init()* method gets the parameter `db_driver` from the `web.xml` file. This parameter specifies which database driver to use and is stored in the configuration of the application.

The *doPost()* method handles the user requests generated by the *POST* method. This method is defined in HttpServlet, which is part of the servlet API.

The *doGet()* method simply call the *doPost()* method to provide a *doGet()* implementation. Hence, this servlet conforms to the servlet API.

The method *debugParameters()* is used for debugging the parameters that are

passed to the servlet. These parameters are printed in the form of an unordered list in the web page that the servlet returns to the client.

The method *handleNormalRequest()* is called from the *doPost()* method if the request from the client is valid and is not a login request.

6.4.5 URLLabel

The `URLLabel` class contains static string variable fields that define the URL rewriting parameter names. When the web application is processing the parameters from a URL request the values are used as labels, which define the parameter keys. These are in the form of *ACTION=parameter*; URL rewriting. URLs are encoded in the web page as links to allow the user to make requests. These requests to the server revolve around an action keyword and various other parameters needed for the action. Each URL on the page must have the values it needs for the request it will make, for example if a link is to navigate to a database table the table name is required.

Information (depending on the view provided by the request) needs to be maintained on the host, database, or table being viewed. The parameter names for these variables are maintained in this class.

6.4.6 URLQueryStringDecoder

The function of the class `URLQueryStringDecoder` is to decode the URL query string into URL parameters. The URL query string is accepted as an argument in its constructor. The decoded parameters can then be obtained by using the *getXXX()* methods on the instance of the class. Internally the class uses a string tokeniser and inner class to pair parameters. The class is declared as final, so that once constructed the values cannot be altered.

The constructor accepts the URL query string as its only argument, as stated previously. It stores the query string internally in the class, in a final string instance, so it is immutable for the lifetime of the classes instance. The con-

structor then calls the method *decode()*.

decode() is a private method used in string decoding. It is responsible for tokenising the query string, using the ampersand character as a delimiter. Resulting from this tokenisation are pairs of *key* with related *values*. The method *decodePair()* is used to tokenise the resulting string containing the *key, values* into separate keys and values. Finally, the method *usePairToSetValues()* is used to store the values against the correct fields in the class.

The *decodePair()* method turns the *key=value* pairs into `Pair` instances (see section 6.4.6). A string tokeniser is used to break the strings up using the delimiter '='.

`Pair` is a private inner class that is only used internally by `URLQueryStringDecoder` to encapsulate *key, value* pairs. It has a constructor and two methods; *getName()* and *getValue()* that are used to access the name (key) and the value of the variable.

The *usePairToSetValues()* method takes an instance of `Pair` and works out from the key (or name) which field the value should be stored under. The stored value can be accessed by using one of the *getXXX()* methods in the class.

getPassword() returns the password as encoded in the URL query string and sent as part of the clients request.

getUser() returns the user as encoded in the URL query string and sent as part of the clients request.

getHost() returns the host as encoded in the URL query string and sent as part of the clients request.

The *getDatabase()* method returns the database as encoded in the URL query string and is sent as part of the clients request.

getTable() returns the name of the table that is being browsed as encoded in the URL query string. It is returned as part of the clients request. Depending on the value of action, this field could also refer to a table that is to be added

or removed.

getAction() returns the action as encoded in the URL query string. This is sent as part of the clients request; it is the action is that the client wishes to do next.

getColumn() returns the column as encoded in the URL query string and returned as part of the clients request.

getColumnType() returns the column type as encoded in the URL query string and is sent as part of the clients request.

getColumnQualifier() returns the column qualifier as encoded in the URL query string and is sent as part of the clients request.

6.4.7 URLQueryStringMaker

The `URLQueryStringMaker` class encodes parameters in URL strings. These parameters are written in the URL after the character '?' as 'name=value' pairs. The constructor accepts various instances as arguments, these are stored using the final keyword, and are therefore immutable. The constructor accepts a base URL, the host, username and password. These are items which do not change for the lifetime of the object. The instance methods in the class are modelled around the form makeUrlString(X,X,X..). These methods all encode the parameters in the URL as a query string. Methods with a differing number of arguments allow for differing amounts of information to be encoded in the URL string. The resulting URLs including their query strings are added to the web pages as anchor-style links, using tag in the form;

$$\langle ahref =$$

The links in the outputted page give users all possible next actions on the database.

6.4.8 SimpleHTMLPage

The `SimpleHTMLPage` class encapsulates the production and merging of HTML output. This class is used by the class `HTMLLogic` to help with the production of the output.

The `SimpleHTMLPage` class has two constructors, each with differing arguments. Both construct a buffer to hold the constructed HTML, construct the headers and footers and set the page title. However, one constructor allows just the page title to be set, while the other constructor provides the option of setting a stylesheet path as well as the page title.

The *merge()* method offers a way for the HTML from another SimpleHTMLPage to be merged with the HTML contained in the instance the method it is called from. The HTML for the body of the page is merged, this is done by calling the *getBodyHTML()* method from the other instance. A boolean variable passed as a parameter allows the inserted content from the other instance to be flagged with text indicating the title of the page it has been merged from.

There are two *println()* methods that are used to insert content to the `SimpleHTMLPage` instance. One accepts a string as its argument, while the other accepts an integer. However, they both use the add method contained within the
`SimpleHTMLPage` class.

The method *setJavaScriptInHeader()* allows any Javascript needed in the head section of the HTML to be inserted. The passed parameter is stored as a string field in this `SimpleHTMLPage` instance.

The method *doHeaders()* is used to construct the HTML for the head section of the web page. It is a private method and is only used internally in the class. The HTML for the head section is built up in a string buffer, a copy of which is returned from this method using the *toString()* method on the `StringBuffer` stored internally. Appended to the `StringBuffer` are the title of the page, the stylesheet used by the page and any Javascript needed in the header.

The method *getTitle()* returns the title of the page as a string

The static method *makeLink()* constructs the HTML for a link using the URL of the link and a string containing the text that is displayed by the link. A HTML anchor element is produced, which is returned as a string.

The method *doFooters()* produces the HTML for closing the web page; namely closing the body and HTML elements with closing tags.

The *add()* method appends a string of HTML to the buffer that builds up the body of the web page.

The method *getBodyHTML()* returns the HTML for the body of the page. This method is used by the method *merge()*.

The *getHTML()* method returns the entire HTML page that is represented by the instance of `SimpleHTMLPage`. This page is produced by concatenating the HTML for the headers, the body and the footer together. The completed HTML is returned as a string.

The method *close()* was added to the `SimpleHTMLPage` class so that it could be used in place of a `PrintStream` instance for debugging purposes. This facility was used to assist in debugging the `mcbrunel` servlet, which is a software component of the LHCb project. When called this method indicates in the outputted web page that, if a `PrintStream` had been used, then it would have been closed.

6.4.9 JDBCSpitfireImpl

The `JDBCSpitfireImpl` class implements the connection layer for JDBC connections. It also provides an implementation of the `SpitfireConnectionLayerBasic` using the JDBC API and a JBDC driver. MySQL provides the database and the actual driver used is MmMysql. Different database drivers can be specified by editing the `web.xml` file. The constructor has a static *createInstance()* method, which is used to create an instance that points to the database host that is to be browsed. The rest of the class deals with implementing the interface.

6.4.10 SOAPSpitfireImpl

The *SOAPSpitfireImpl* class is a work in progress and is, at present, not fully implemented. The class should implement the connection layer using SOAP connections for the web application. The class `SOAPSpitfireImpl` implements the interface `SpitfireConnectionBasic`, meaning that it can be interchanged with `JDBCSpitfireImpl`. `JDBCSpitfireImpl` provides a reference implementation of the functionality that is to be provided. The implementation of the `SOAPSpitfireImpl` relies on the functionality defined in the following 3 interfaces;-

- `org.edg.data.spitfire.service.SpitfireAdmin`
- `org.edg.data.spitfire.service.SpitfireCore`
- `org.edg.data.spitfire.service.SpitfireInfo`.

Therefore, Spitfire SOAP system provides the same services as the JDBC implementation.

6.5 Deployment

For ease of deployment the Spitfire Browser was packaged as a web application. The build tool, ANT (see section 3.3.7) , was used to build this web application. This involved the following steps;-

- Creating a destination directory structure for the web application
- Compiling the Java code
- Copying all the required files into the web application file structure
- Packaging the files and directory structure as a WAR file.

In the ANT script, an XML file, the actions to be performed were placed as XML elements.

An ANT script is an XML document which has, amongst other elements, a root element `'project'`, and `property` elements that define properties and `targets` which contain the scripts which can be invoked.

The ANT script for SpitfireWeb defined these properties;-

- *build* - the root of where the web application should be built (typically in the webapps directory of the Tomcat (see [59]) installation for convenience during the project).

- *webroot* - the root of the web application, a directory in which the webapp is contained. The webroot is itself contained in the webapps directory, where the other web application deployed on Tomcat will be contained.

- *classfiles*, a location defined relatively from webroot where the class files should be copied. Something like `../WEB-INF/classes`.

- *source*, the location of the source code for the webapp.

The ANT script for SpitfireWeb defined two targets;-

- *init*, for creating the directories where the web app will be build.

- *build*, for compiling the webapp and copying the extra resource files to the webapp folder.

6.6 Summary

The Spitfire Admin Browser was tested using MySQL database and mm MySQL database driver. JDBC implementations of the connection layer were tested and found fully functional; databases, database tables and columns were created, dropped, modified, etc. The user interface was found to be relatively simple to use and understand. Deployment was via the ANT scripts written to compile the Web Application. The Spitfire Admin Browser was not used in production.

6.7 Possible Progression

Spitfire Web as presented here represented a proof-of-concept web application. One short coming was the lack of a finished SOAP implementation for the web application to use. Completing the implementation of this would an obvious progression. A SOAP server to test this implementation would be required. Another possible progression to the system would be the addition of user roles allowing for various privileges. For example, this would mean that only correctly authorised users could drop database tables.

Chapter 7

Monte-Carlo Array Processor

7.1 Introduction

General

The *Monte-Carlo Array Processor* (MAP) built in 1998 by the University of Liverpool is a commodity supercomputer. The key concepts of MAP are the use of *Commodity Off The Shelf Hardware* (COTS), having many nodes, typically hundreds, and a custom software setup, written in house at Liverpool. This software allows MAP to work in a unique fashion enabling simulations to be made massively parallel.

Hardware

The MAP hardware consists of nodes on which the simulations are run. These nodes are commodity PCs mounted in racks. Each of these nodes is connected into MAPs **private** *local area network* (LAN). 'COMPASS' nodes (servers) control MAP, provide a gateway to the outside world (the *wide area network* or WAN) and act as staging for the data output by the nodes. The *COMPASS* nodes are more highly specified than the other 'simulation' nodes.

Software

My work has involved developing software that provides web and grid interfaces to the original MAP system by interfacing with the MAP core software. The software MAP uses for controlling the array is custom written. As discussed above, MAP consists of nodes, for running jobs and COMPASS nodes that act as controllers and gateways to the system.

When running a job, MAP runs one simulation per node. To do this a system of controlling and monitoring the nodes is required. This system has two main parts; software that runs on the COMPASS systems and software that runs on the nodes. For ease of maintenance and reliability the software on each MAP node is kept minimal. The operating system is a RedHat flavour of Linux, on top of this runs the custom MAP software.

The method used to run jobs on MAP is distinctly different to the method used to run jobs on a PBS system (see PBS, page 52). A single node runs a job by having an entire system (operating system, libraries, application and required files) installed on it. This is repeated for each node in the cluster simultaneously. Each time a job is run on the cluster this procedure is repeated. Jobs tend to run for periods of many hours, so the time taken to set up the jobs on all the nodes is small in comparison to the run time of the job.

When it is decided a simulation will be run on MAP a job is set up on a single machine. This is usually the machine of the person setting up the simulation. Once the job is working for this single machine, it is effectively cloned to the MAP nodes, resulting in the job running in a embarrassingly parallel fashion. Differences between the simulations are achieved by giving different input files to each machine or specifying starting parameters that are different for each machine. Cloning of the job to all the nodes is achieved by the custom software.

Usage

Refining the LHCb[13] vertex detector design was one of the main incentives for MAP, although its subsequent usage has been more general. MAP can be

used to solve many physical, scientific and engineering problems.

Overview

Figure 7.1 illustrates the MAP system. On the far left is the user with a client. The client is typically a web browser, a command line executable or an *X-Windows* program. MAP consists of nodes in a private network, on which the jobs are run (lower right of figure) and COMPASS nodes control the system. On the COMPASS nodes is software to accept incoming communication (for example the request to submit a job) and software that controls the nodes. The DUCK system (centre of figure, see section 8) web-enables the MAP system. More information about MAP is contained in the following paper [60].

7.2 Hardware

7.2.1 General

There are two generations of the MAP hardware at the University of Liverpool, MAP 1 and MAP 2. Whilst MAP 2 uses more modern hardware than MAP 1 and also has a greater number of nodes, the concepts behind both MAP systems are the same. The MAP hardware shares some concepts with the computer clusters discussed in section 5.1.2 but also has some unique features discussed here.

The nodes, as stated previously, are commodity PCs. Typically in the order of hundreds of nodes are used and they are all identical in terms of hardware. Inside the rack-mountable box, a MAP node will have standard PC components;-

- A motherboard
- A central processor unit
- A local fixed disk
- I/O devices for removable media, usually floppy disk drive and CD-ROM drive.
- A power supply with cooling
- A network card
- A front mounted panel with various ATX sockets on it, usually keyboard, mouse, VDU, serial ports and network sockets.

The nodes are mounted in racks, with each rack holding approximately 30 nodes, depending on the MAP generation. Sufficient cooling must be made available to dissipate the combined heat output of the nodes. The racks must also facilitate easy removal of the nodes for servicing and repair.

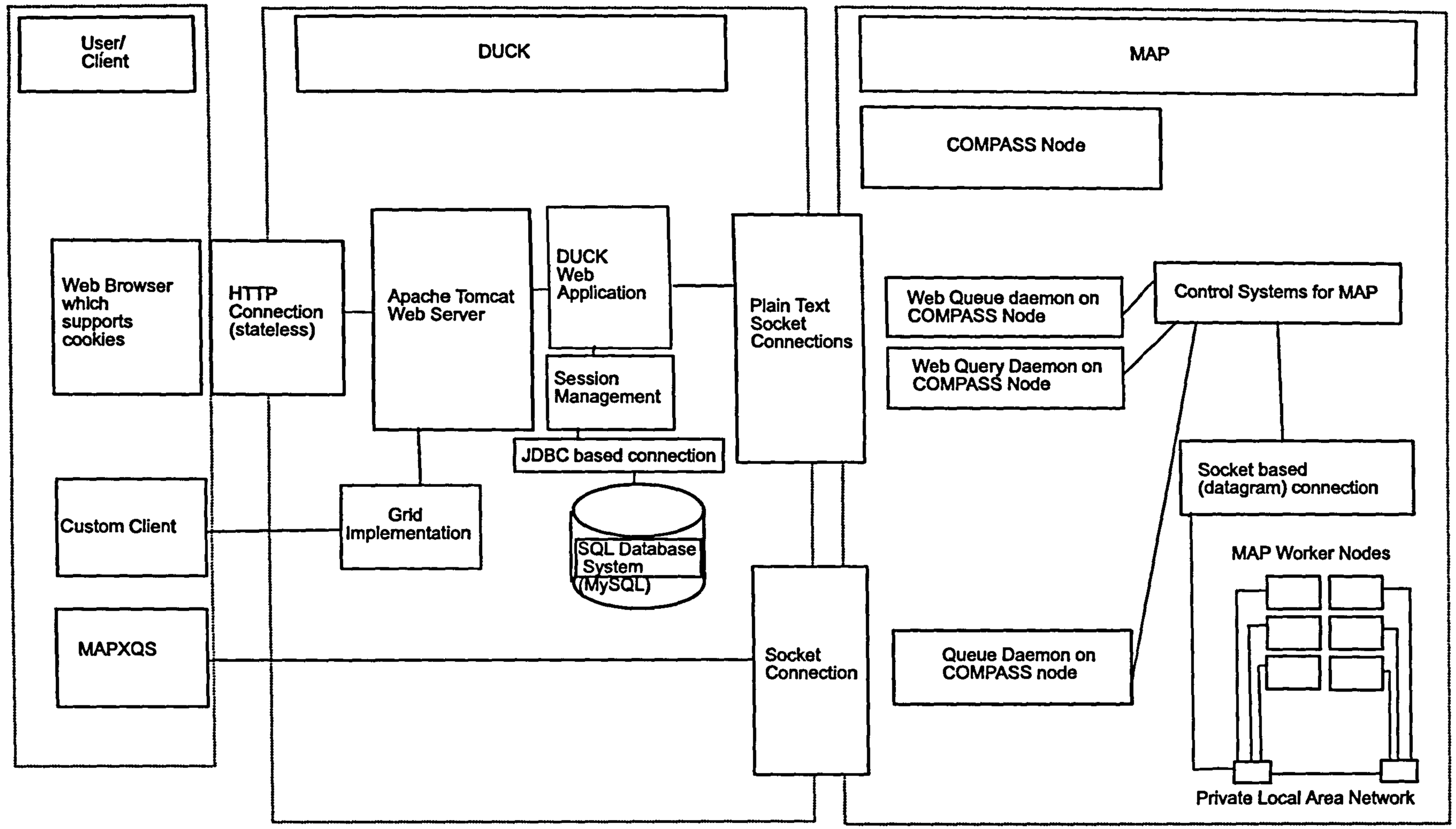

Figure 7.1: An overview of the MAP system, also including the DUCK (Dedolved User Control Console).

The nodes and COMPASS nodes are interconnected on a private network. The network technology chosen for MAP was ethernet. Each node has an ethernet adapter card, allowing it to be connected into ethernet switches. These in turn are connected to the COMPASS nodes. MAP can be configured to form separate disconnected networks, meaning it can be partitioned into groups of nodes, each of which act like a complete, separate MAP system.

Nodes are numbered sequentially in the racks. The IP address of each node is static and in the range allocated for private internets[61]. Hence, the IP address for each node follows the form of 192.168.x.x, where x.x represents the IP subnet of each individual node. These IP addresses relate to the physical position of nodes in the racks.

The COMPASS nodes act as a gateway to the outside world, the WAN. Each COMPASS node has two network cards, one for the connection to the MAP private network and the other for connecting to the outside world. This arrangement means the private network, and hence the nodes, are not visible to the outside world (internet, LAN). Therefore, they are not exposed to same security issues as a machine directly connected to an internet or LAN would be. However, they are able to see the outside world via *network address translation* (NAT). This allows the nodes to access remote resources such as databases. The COMPASS nodes are perhaps better equipped to handle I/O operations; they have more powerful processors and are equipped with a relatively large amount of disk space.

7.2.2 MAP 1

MAP 1 is the first generation of MAP. It consists of 300 nodes and 6 COMPASS nodes. The nodes have the following specification;-

- 400MHz Intel Pentium II processors
- 100 BaseT ethernet cards
- 20Gb IDE fixed disk.

The nodes are grouped into racks of 30 machines, with each rack housing, in addition to the nodes, two16 port 100 baseT *ethernet switches.* Nodes are wired into these switches, which in turn are connected to the COMPASS nodes.

The COMPASS nodes have the following specification; -

- 700MHz Intel Pentium III processor
- 10 SCSI disks, 50Gb each, giving 500 Gb of storage per COMPASS node
- Two ethernet cards.

7.2.3 MAP 2

MAP 2 is the second generation of MAP with an increased number of nodes and more up-to-date hardware. It is comprised of 940 nodes and multiple front end nodes. MAP 2 tends to be used for more generalised cluster computing.

The nodes have the following specification;-

- 3Ghz Intel Pentium 4 processor, dual CPU core - two CPUs in one chip.
- Gigabit BaseT ethernet, with a Force 10 E600 gigabit switch.
- 120Gb IDE fixed disk.

The nodes are grouped into racks of 38 machines, with each rack housing, in addition to the nodes, two 24 port gigabit baseT ethernet switches. These switches are then connected to the front ends, which vary in specification.

The nodes have a relatively large heat output that needs dissipating, with each rack generating $6.5\ KW$ of heat. Most of the racks have air conditioning for cooling, whilst the remainder have their air circulation cooled by water cooled air conditioners.

7.3 Key Features

Hardware is only one aspect of MAP. There are some key concepts and features that are unique to MAP and allow it to perform its work. These features are discussed in the following section.

7.3.1 Scripted Boot

The array uses a scripted boot system. This is essential to the way MAP works; without it MAP would not function as intended. The scripted boot provides a mechanism which falls back to a working system if an error occurs in the boot process.

The common Linux *boot loaders* (including *Grub*[62] and *LILO*[63]) do not perform a scripted boot in the fashion required for the array to work. MAP employs *LoadLIN*[64] in combination with a DOS (Disk Operating System - the precursor to Windows). The DOS system batch file `autoexec.bat` contains a set of commands that are executed at startup time. This file is used to provide the required scripting. For a machine with a working operating system on which we wish to install a new system the following process is used.

The new system is installed on to a new partition and the boot sequence is altered to boot into the new partition. As the machine is booted the configuration is changed (hence a need for write access at startup time). The old system (named `newsys.linux`) is renamed to `oldsys.linux`. Then the `oldsys.linux` is booted (this is always booted). A new system is only booted when a `newsys.linux` exists. The scripted boot is essential to the way in which MAP works, allowing it to revert to a working default if the system fails to boot.

In the future PXE boot system could be used[1]. This is where the boot process is driven over the network. The boot sequence is driven by a file on a centralised server. One of the COMPASS nodes or front ends could be used for this.

[1]In fact in MAP2, PXE boot is currently used.

7.3.2 Transmitting Data

The nodes and COMPASS nodes in the array are connected by a private ethernet network, over which jobs are set up.

To transmit files a protocol based on *User Datagram Protocol* (UDP) is used. A potential alternative is TCP/IP (see 5.2.1), which is also a member of the IP family of protocols. UDP is a connectionless protocol that runs on IP networks, with very few error recovery services. The use of UDP means that re-requesting lost packets of data is not handled by the transport software; instead the application has to handled this. UDP was chosen as the basis for the protocol used for communication inside MAP.

UDP was a prime choice for several reasons, firstly because it can transmit in *broadcast mode*; TCP/IP cannot. Broadcast transmissions are needed by MAP so that nodes can simultaneously receive the same file, in contrast to it needing to be transferred individually to each node. This approach would produce a lot more network overhead.

Secondly UDP has a low overhead, meaning there is less load on the network. This low overhead is partly due to the lack of error checking. Whilst the TCP/IP stack has sophisticated error checking and guaranteed delivery of data, UDP has no such guarantees regarding to data reception. A UDP packet can be lost, or packets can arrive in any order. This is very much in contrast to TCP/IPs guaranteed delivery and ordering of packets. The private network connecting the array is relatively reliable compared to the internet or a public LAN; rendering the lack of error checking and correction mechanisms acceptable and hence UDPs suitability as a protocol for MAP internal communication.

Internal communication between COMPASS nodes and nodes in the MAP system works as described below. This software was written for the MAP system.

Data (usually files) is transmitted from COMPASS nodes to nodes via the private network, in the form of UDP broadcast packets. As the transfer of data

proceeds each node keeps a record of which packets they have not received and when asked by the COMPASS node, the nodes inform the COMPASS node which packets, if any, they are missing. The packets are then re-transmitted to the machines that need them.

The number of packets broadcast in one communication is called the window size. The window size is dependent on quality of line, which because we are running on a private network, has a good degree of reliability.

After the correct number of packets have been broadcast, the COMPASS node needs to send the packets that were not received by the nodes again. It transmits the packets that were lost to the machines that need them. The functionality described above is provided by the MAP software and not the UDP system.

A crucial point of the MAP nodes network interaction is their network silence. The nodes are silent on the network until information is requested from them. Only if a request is made for the node to provide information does the node transmit data on to the network.

If the nodes broadcast information by their 'own' accord the private network would quickly become saturated with network traffic and break down. For a small number of nodes, less than say 100 (usually modern network technology) this is not a problem, but with large numbers of nodes generating network traffic, it does become an issue.

A node inspecting a packet looks at the first 32 bit block in the packet. Using this block the node can determine whether the packet is intended for it. If the packet is not intended for the node, it is discarded.

The numbering scheme from MAP nodes was discussed in section 7.2.1. Encoded in the 32 bit block is the address of the first machine that needs to receive the data. The block then specifies the number of subsequent machines after this block that need this data.

7.3.3 Fixed Disk Partitioning

Central to the way in which the MAP software runs is the *partitioning* [2] of the fixed disks (or hard disks) on each node.

The original procedure for installing the MAP software on to the nodes was to clone a single fixed disk. One fixed disk was partitioned into the required partitions using a disk partitioning tool and then set up with the required files for the basic system. This fixed disk was cloned using a disk cloning tool. Therefore all nodes start with an identical partition and file set. All of the partitions on the fixed disks are primary partitions. The 4 partitions are; the *DOS/Boot partition*, the *run partition*, the *swap partition* and the *control partition.*

The *DOS (Disk Operating System)/boot partition* is used to boot the system. It takes up a small amount of space on the nodes fixed disk, approximately 50MB.

The *run partition* is the partition where the job is run. It occupies the majority of the disk space of the node's fixed disk (approximately 18.5GB).

Basic Linux is used for the running partition; meaning only core packages are installed. Once a job runs on a single machine, the system and job are 'tarred up' to a tar file. When a job is run on MAP this tar file[3] is un-tarred to this partition on every node. The run partition operating system is a cut down version of Linux.

The process for making the job tar file is described as follows, where 'desktop' refers to the machine the job has been setup on. The OS (files, libraries, etc.) from 'desktop' machine is 'tarred up' into a tar ball archive.

---

[2]Partitioning[65] is a mechanism used to divide up a hard disk into *partitions.* These partitions appear to the software like physically separate hard disks. A hard disk can be divided into a maximum of 4 primary partitions. Any further division is done by using *logical drives* inside *extended partitions.*

[3]A *Tar Ball* is a collection of files compressed into a single file. The process of making a tar ball is known as 'tarring up' files.

The kernel source code of the 'desktop' machines kernel is needed to build a kernel that works for a MAP node. A kernel may work for a given desktop machine, but it is highly probable that the MAP node hardware is different and the kernel, therefore, needs to be altered and re-compiled to get it working on MAP hardware. Typical differences are associated with network cards.

The *swap partition* provides swap space for the node needed by Linux. This is a partition of 400MB size, which is a little more than the physical memory in each node.

The *control partition* is a 1GB sized partition on each nodes fixed disk. Setting up the run partition is the function of this partition. It contains the 'real' operating system for the node. Currently RedHat Linux release 9 is used. This version of RedHat is cut down and optimised for setting up jobs and running the network for transferring files (broadcast file transfer). In addition, the operating system kernel is re-written to be optimised for the MAP hardware. The operating system is much simplified and specialised for peer to peer, file transfer and control commands for the array. Files are transferred by broadcast mode on setup. The MAP sender and receiver send and receive one megabyte at a time. The system is very similar to *File Transfer Protocol* (FTP).

The operating system and software on this partition knows how to construct the *run partition.* Using Linux means that there is control over which partitions are mounted at run time. The control partition is not mounted when the job is running, meaning it cannot be affected by the job or its actions. This protects the partition as a fall-safe system to allow recovery of nodes when jobs fail.

7.4 Core Software Description

7.4.1 General

The software that runs MAP has been developed at The University of Liverpool. The core software for operating the array consists of a number of C

programs and number of C daemons[4]. There are a number of data structures that are central to the system. These are encapsulated using C structures, `structs`. One of particular importance is the Job Control Block, which encapsulates data relating to a computing job. This data tends to be related to the computing side of the job and not to do with the simulation/run (for example the Job Control Block does not encapsulate any physics or engineering data).

7.4.2 Job Control Block

The Job Control Block is modelled in MAP software as a C structure. In the core MAP software the Job Control Block is written to disk to allow it to be retrieved by other invocations of the software. The parameters and their functions are discussed further in the section .

7.4.3 Queue Daemon

The *MAP Queue daemon* is an entry point to the core MAP software. It is a very short program that runs as a *daemon.* The *Local Control Daemon* software (see section 7.4.5) performs various checks and tasks for MAP Queue Daemon. The MAP Queue daemon sits listening on a network socket for a client to submit a job. The Queue daemon can be bound to any valid available port. There are two versions of the MAP Queue daemon, an older daemon designed to work with MAPXQS and a newer daemon designed to work with the Java based DUCK system. The latter is known as the *web queue daemon* to distinguish it from the previous version of the queue daemon.

Both queue daemons can check that the client connecting is on a list of allowed clients. This list is in the file `clients`. The MAP function *allowed_client()* checks a client against this list.

[4]A daemon is a program that runs constantly. It is launched by a process and then detaches itself from its parent process.

7.4.4 Query Daemon

The *query daemon* is for exchanging information between MAPs core software, mainly the queue, to outer tiers of the software; the DUCK system. This daemon provides information that allows DUCK to display the status of jobs submitted to MAP.

7.4.5 Local Control Daemon

The *local control daemon* is installed on every COMPASS node. It is invoked by a queue daemon; when this occurs local control daemon runs MAP Run Job or *mrunjob*. This is achieved by using a generic *spawn* or *fork()*. The fork or spawn is where a child process is launched from a parent. The parent process then exits, leaving the child process still running.

The local control daemon also performs checks such as whether there is enough disk space to run a job and various checks related to the MAP cluster.

7.4.6 MAP Run Job

MAP run job is spawned by the local control daemon and runs on the COMPASS nodes. The program queries each MAP node in turn. MAP run job talks to instances of node control daemon when jobs are in the process of being set up. A copy of the node control daemon resides on each node, with the task of setting up the job on that node.

MAP run job is also responsible for concurrently copying back the output to the COMPASS nodes when jobs are running.

The following software structures and variables are used by MAP Run Job when communicating data. The *node control block status block*, the *query status* command, the *node state*, which is communicated through a structure, the *nd status* (resides on the node; MAP NCD keeps this status up to date) and the *cnct* structure.

The node control daemon also has a control function for the nodes. Valid commands for this are; `RESET`, `SHUTDOWN`, `BOUNCE` - shutdowns and restarts the node quickly, `SHUTDOWNS` - stops the node, `SHUTDOWNR` reboots the node, `SHUTDOWNRF` (fast reboot of the node, no checks), `SEND STATUS` - sends status of the node, `WAKE` - wakes the node and `SLEEP` - puts the node to sleep.

Errors occurring that stop program executing can cause problems for the cluster as a whole. The COMPASS node will continue to receive output from a sender in an error state and just discard it. This solution is less problematic. A segmentation fault in a program puts the node into sleep state. The `EXEC` remote shell is available for the cluster. Also available is PP-FILE, which is effectively *File Transfer Protocol* (FTP).

There are 3 administrative commands for the cluster as a whole; `startmap`, which starts MAP, `stopmap`, which stops MAP and `resetmap`, which resets MAP.

7.4.7 MAP NCD

MAP Node Control Daemon (MAP NCD) is a piece of daemon software that runs on every node in the array. The daemon listens on 3 network sockets. All jobs are run as child processes of MAP NCD. Hence MAP NCD, being the parent of the job, knows when the job has finished on each node. This information is available to MAP NCD because the operating system gives a parent process the status of its child processes.

MAP NCD controls the data output from jobs on each node. As a job runs on each node output is built up on the nodes fixed disk. MAP NCD wakes up approximately every minute and sends the differences in the output files to the server. The server receives hundreds of versions of the same file generated by each node, it needs to be told how to differentiate between versions of the same files from different nodes. This is done by using the machine name of the node.

The output provides a 'heartbeat' for each node; whilst output is being pro-

duced on a given node the job is running on that node. When no more output is being produced by the node the job is finished. Once the job has finished output is available from the COMPASS nodes.

MAP NCD is installed on the run partition (see section 7.3.3). MAP NCD is statically linked. [5]

MAP NCD is updated using the following process. MAP NCD listens for an updated version of itself. When an updated version is available it receives the new version. MAP NCD then renames itself and then shuts down. There is a mechanism to keep MAP NCD running, which is *init()*, so this mechanism automatically starts the new version of MAP NCD.

7.5 X-Windows Interface to MAP

The *MAPXQS client* is an X-Windows based client for the MAP array. It allows users to use X-windows controls to set up and submit MAP jobs. The program was retained as the method to set up MAP jobs for expert users.

7.6 Summary

This chapter introduces MAP, both in terms of the hardware of the cluster and also its custom software setup. The setup of the hardware, in terms of installation and network connects was discussed. Key software features of the system were highlighted. This chapter provides the grounding for the next chapter which discusses DUCK; a web interface to MAP.

---

[5]Static linking is where any libraries needed by the executable are compiled into the executable, hence problems with different versions of libraries on different machines are avoided. *Dynamic linking* is where libraries are found dynamically at runtime. By using dynamic linking executables can share libraries and hence take up less space.

Chapter 8

A Web based Job Submission Tool for MAP

8.1 Introduction

The *Devolved User Control Konsole* ('DUCK') was developed as an outer tier to MAP. Central to the motivation for producing DUCK was to provide a simple, non-MAP-expert method, for users to submit jobs. The system for submitting MAP jobs discussed in section 7.5 is difficult for non-experts of MAP to use.

A large amount of the work revolved around the development of an interface between DUCK and the MAP core software. Once this was finalised, then work was done to develop the DUCK tier of the software further, with a web interface and some steps toward a grid interface.

The technologies used in DUCK were *Java* (see section 3.3.6), *SQL databases* (see 4.7.2)and *socket-based networking* (see 5.2.2). Like the Spitfire Admin Browser (see section 6) servlets were used. Unlike the Spitfire Admin Browser, multiple servlets were used instead of just a single servlet. Having a web interface means that the user only needs the correct authorisation and any reasonably modern web browser to submit a job that has been set up on MAP.

The completed DUCK package sits as a tier in a tiered model. The MAP array, its private network and COMPASS nodes form the *first tier*. DUCK itself forms the *second tier* and the clients web browser forms the *third tier*.

Readers may wish to read "Starting a Job using DUCK" (section 8.6) first and use the cross-references contained in it to further read the relevant sections as they read that section.

8.2 Requirements

The requirements for the system were;-

- To have a system that enables *non-experts* can easily run jobs on MAP, changing some parameters if required
- To explore the concepts involved in turning MAP into a *grid resource*.
- To provide a system allowing users can check the *job queue* for MAP.

8.3 Development Process

8.3.1 Overview

There were a large number of challenges in the development of the DUCK system. Primarily, there was the problem of communication with MAP, effectively the first tier. This came down to the method used to 'link' the second tier to the first tier. The possible solutions, tested solutions and final solution are discussed in this section.

Another problem was the formatting of the information exchange once the connection system had been established.

The implementation of the client, the third tier, presented many options. Again, these are described in this section along with the final solution which was a web interface.

One of the constant considerations was 'perfection' versus getting a working implementation in a realistic time scale. There were many times when pieces of code were rewritten/refactored at a later date as the software and requirements evolved.

8.3.2 Communication to MAP

The challenge of communication to MAP was the first problem tackled. The COMPASS servers act as a gateway to the MAP array; thus this is the first point of communication to MAP. In the tiered model we are effectively connecting the first and second tiers of the overall system (although it could be argued that the array is the first tier and MAP is the second; but I'm referring to MAP as a complete entity).

One initial decision was where the software itself should sit, either on the COMPASS node or on a separate server. The latter situation would sit well in the tiered model system and devolves the system more.

JNI approach

The *Java Native Interface* (JNI) is a programming interface for accessing native code from Java and vice-versa. This means that platform-specific code, meaning non-Java code such as C, C++ and Fortran, can be utilitised by Java and Java routines used by native code. So, for example, a Java program could access portions of C code outside the Java virtual machine (JVM). Another example would be a native C program accessing Java code running on a JVM in an embedded fashion. Another potential solution was CORBA (see I.2), which was deemed to add too much complexity to the solution.

RMI approach

Remote Method Invocation (RMI) (see appendix I.5) provides a simple way to do remote computing with Java. This was considered as a potential solution.

It would have involved running an RMI server on a COMPASS node and performing RMI transactions between MAP and DUCK.

Socket approach

The socket approach is the simplest and most basic way of communication over a network. A server listens on a server socket for a client to connect. Once a client has connected data can be passed backwards and forwards down the socket. The socket abstraction dates from the 1970's and sits on top of TCP/IP in the case of Java. A more detailed description of this technology is given in section 5.2.2.

8.3.3 Initial Program

A very simple initial program was written to establish communication with the MAP web query daemon. Writing this allowed problems in the socket communications to be overcome. One particular problem was deciding on a specific line terminator character; which needs to be the same at both ends of the software.

8.3.4 Persistency

HTTP The Stateless Protocol

A web browser uses *Hyper Text Transfer Protocol* (HTTP), which is a stateless protocol. This means that information between transactions is not retained by HTTP. A client will typically request a webpage, which is returned by the server. When the client requests another webpage, even if it is the next in a series, the server sees this request as completely separate and unrelated. This causes problems for applications that need to maintain information between pages. A commonly used example is a *shopping basket* at an e-commerce site. Here, products put into the basket need to be remembered as the user browses round the rest of the website. The different ways to get round this limitation of

HTTP are discussed in the following section. Typically the task of maintaining information between web page requests is called *session tracking.*

Client-side Cookies

Cookies are a persistency mechanism that is relatively well known; mainly because of media attention to privacy issues involved with them.

Cookies are *[key, value]* pairs. For each key there is a corresponding value. They are set by the server and stored on the client as a series of small text files (for each cookie) or grouped together in a single text file. Cookies originate from the server, which can request a browser to set a cookie. If the name of the cookie (the key) already exists, it is over written with the new value. Cookies are sent as instructions in the HTTP header section. Once a cookie is sent the client sends it back with every request to the server that sent it. Using the above method information can be made persistent between pages.

The privacy issues involving cookies mean that they can be used to track a user's movements on a web site. As a result some people may switch off cookies in their browser program.

Although cookies are not used to store information on the client in the DUCK system, a single cookie is used to store an ID that links the client to stored information on the server. Hence, the client must be able to accept and return cookies to use the DUCK web interface.

Hidden Forms

In HTML documents a section called a form can be used to get input from users. Elements called controls are contained in the form to allow users to complete the form. Forms can also be used to maintain information between page requests. A form that is not rendered on the page is used; this is known as a *hidden form.* When a form is submitted the information in it is sent to the server. Using this technique persistency can be provided between requests. This technique, although more secure than URL-rewriting (see below), is not

completely secure. The information is passed as plain text on a non-encrypted connection; this leads to potential eavesdropping of the data. Furthermore, the user can 'view source' and see the hidden form and its values in the source code.

URL re-writing

URL re-writing involves storing parameters in the URL. The HTTP get request is made up of the location of the resource and an optional query string. This query string can contain *[key, value]* pairs. It is a solution that can work when cookies will not. It also moves the 'storage' to the client (like hidden forms and cookies).

There are several disadvantages to this technique. Firstly, there can be environmental limitations on the length of the URL. This would be a problem if moderate amounts of data were being stored. Secondly, there are significant privacy issues; the data is visible in the browsers address bar. This means the user himself/herself can see the data, not to mention people viewing the screen over the users shoulder. Thirdly, the data can be seen easily in browser history records and during transmission. Finally, from a technical point of view, implementation becomes more difficult as writing links to other pages is a longer process. It has to be done dynamically, as the parameters need encoding in each link on the page. The resulting links do not bookmark well in the users browser.

Server-side Session Management

HttpSessions are a solution provided by the Java servlet API. The `javax.servlet.http.HttpSession` interface provides a generalised means of persistency. The web server provides the session management as defined in the interface. Each web-server (for example Apache Tomcat[66], Oracle application server[67]) may implement the interface in a different fashion using different means. The most common implementations use cookies and/or URL

re-writing. URL re-writing tends to be used when cookies cannot be set because of client setting or firewalls that filter out cookie information. These common implementations suffer from the same limitations discussed earlier in URL re-writing and cookies (see sections 8.3.4 and 8.3.4). The advantage of server-side session management is that the interface for the persistency is standardised and ensures the portability of the web application. One far-sighted disadvantage of session-management is that techniques almost exclusively use web-based means, making it less suitable for grid-like application.

Server-side Database

The solution decided upon was a client-server system with the persistent information being stored on the server in a database. This means the server takes the load of storing the information. A single cookie, stored on the client machine used as an ID, was used on the client side to associate client requests with the correct data. This cookie links the client to information stored on the server, which is stored in the form of database tables. Connectivity to the database is provided via *Java DataBase Connectivity* (JDBC), see 4.7.4.

There are numerous advantages of the client-server solution. Firstly, only information relevant for the request is passed between client and server, this means that all the data is not passed back and forwards constantly. This protects sensitive information. Secondly, there is less limitation on the amount of information that can be exchanged between the server and the client, which was a problem with other methods. Thirdly, the systems on the server side can be reused to build grid aware systems. Finally, the information stored in the database can be used for other purposes aside from persistency between HTTP requests. Such purposes include logging and diagnostics.

The disadvantages of such a system are that more work is required to implement the system and it is a more complex system, involving a database.

MySQL was used to provide the database. It is an open source *relational database.* JDBC (see 4.7.4) was used to connect to the database from the web

application. The database driver used was a type 4 database driver (see 4.7.4), meaning that it is implemented in pure Java and issues requests directly to the database.

Another potential solution was to store persistent data as XML in a native XML database such as Xindice[45]. This would have added complexity, by requiring persistent data to be in the form of XML, but would have perhaps eased development of the web services prototypes.

8.3.5 Problem of handling files

The files needed to set up a MAP job are large, in the order of gigabytes, therefore it was deemed impractical to upload these files using a web browser.

If a Java application was used as the client, then files could be uploaded from the client. Using an applet as the client leads to users having to make security allowances to allow the applet to access their file system.

The solution decided upon used a web browser. It was decided that the MAP XQS system (see 7.5) would be retained for an expert in the MAP system to set up jobs. Users could then easily run the setup jobs with different parameters from the web browser based client. See 8.3.9 to read more about the client choices.

8.3.6 Job Control Block Modelling

The Job Control Block parameters deemed necessary for the DUCK side of MAP were;-

- *account name*, the MAP account name. MAP accounts consist of groupings of users with valid sites and a disk space limit
- *saved job name*, the name of the job to be run. Previously known as the executable name. This is a job that has been set up previously.

- *site*, this is the site the user is submitting the job from. The site has to be listed in the account data for it to be allowed. It is passed as a numeric IP address.

- *user*, sets the user. The user has to be a member of the account it is using.

- *disk space.* How much disk space the user will be using locally on the MAP cluster. If they are using a remote location this is of no concern to the MAP system.

- *tar file.* The name of the tar file used in the job.

- *over write.* Whether the output should overwrite a previous run of the job. (true or false)

- *random first.* The first random number, an integer, used as a seed for jobs. Values from 1 to 9 inclusive. Cannot be zero. This seed is used to introduce differences between the jobs running on different nodes.

- *registered days.* The number of days the job is registered to run on MAP. Cannot be a negative number.

- *end script.* The filenames of the endscripts to run when the job has finished

- *job name.* The name of the job.

- *edit files.* The filenames of the edit files

- *multi files.* The filenames of the multifiles. There must be enough multifiles for each machine. Multifiles are used to introduce differences between nodes. Each node receives a different multifile, meaning different output is produced by each node instead of having, for example, 300 identical sets of output.

- *split files.* The filenames of the split files. This is a similar idea to the multifile, except a single file is used and split up. Each node receives a

portion of the split file. The file can also contain common instructions for every node in the cluster.

- *return files.* The filenames of the return files. These are the files that contain the output from each node.
- *exclude files.* The filenames of the exclude files, files which shouldn't be copied to the nodes.
- *NFS mounts.* Any NFS mounts that need to be available when the job is running
- *registry text.* Text to be stored in the registry system on MAP
- *job directory.* The job directory. The directory path in which the entire job resides, including the executable and related files.
- *run time.* The length of the runtime in hours, this is the time the job will run for. It is allowed to be a decimal number.
- *queue.* The queue name, this is largely redundant as there is only one queue on the core MAP software.
- *random seed.* The random seed, a whole number. This value is used a the starting seed for the job. It is the same value for each node.
- *Linux version.* The Linux version to use for the job, a continuous string specifying the version, i.e. "RH6.2"
- *kill percentage.* The number of nodes that have completed the job before the job can be killed. This exists because a small number of jobs will run for a much greater time than the other jobs. The need for this functionality was discovered in testing the system.
- *kill time.* The time in seconds after which a job can be killed.
- *remote output.* A remote directory for output. This can be left blank for if the output is local (meaning it goes to a COMPASS node at Liverpool).

The remote directory would typically be a valid directory somewhere like CERN. There needs to exist a valid listener for the directory.

- *fetch sync.* Whether fetch sync is used (*true* or *false*). The fetch sync setting refers to whether the output is moved to the COMPASS node as it is created, as the job is running or it is only copied to the COMPASS node when the job has completed. Fetch sync needs to be off if the simulation uses its own output for some reason during the simulation.

- *staggered start.* The staggered start delay in seconds. This is required if the simulation accesses databases or similar resources. The staggered start stops a high number of concurrent requests that could cause problems.

- *number of processors.* For supplying (to the web application) the number of processors available on MAP.

- *ignore warnings.* Relevant when a submitted job has warnings. Allows the user to submit the job regardless of warnings existing. (true or false)

- *Session ID*, used mainly by DUCK to link Job Control Blocks to a user and session.

The software was arranged into packages. Using packages in Java is a similar concept to namespaces in C++ and many other languages. The packaging structure allows different components (classes) of the software to be arranged in logical groupings. Packaging also prevents clashes in class and method naming. At first the packaging started with only packaged defined, as development proceeded the number of packages grew.

`duck.JobControlImpl` is an implementation of the JobControl interface. The class implements all the get and set methods defined in the JobControl interface. The parameters are stored internally as immutable strings (all strings are immutable in Java). The `JobControlImpl` can be obtained as a JobControl interface by the method *getState()*. This means that the JobControlImpl can be used (with the stored values) in situations that require the use of the

interface JobContol. There are various static methods that return Job Control Blocks filled with various sets of parameters;-

- An example Job Control Block with typical values
- A blank Job Control Block with empty string for all the parameters
- A Job Control Block with parameters set as the names of the parameters.
- A Job Control Block with parameters set as randomised strings

The class implements methods that can perform basic checks on the data.

8.3.7 Job Validity Checking

The parameters in the Job Control Block are checked for their validity. Checking occurs in the DUCK tier and on the MAP tier.

8.3.8 Classification of Feedback

The core MAP software provides feedback about MAP jobs classified in 3 levels of severity. These levels were decided in the development of the software and are described below.

- An error is defined as a problem with a job that means the job cannot run.
- A warning is defined as a problem with a job that does not stop it running, but could cause problems. If this occurs the user is given the option of running or cancelling the job.
- Information is a category of feedback that informs the user of the something. It does not stop the job running.

8.3.9 Client

The web browser was decided upon as the client for the DUCK system. This provides a very flexible system because the web browser is a very common piece of software.

A possible solution to providing a client for users was to write a Java application. Having a Java application as a client gives more functionality than, for example, a web browser. The Java platform is platform independent meaning the application should be deployable on most modern computer platforms. One disadvantage is that the application will need installing (along with Java if this has not been previously installed). The application approach puts greater demands on the resources of the host machine, but takes some of the load off the server side. This could be a problem if the client side machine is relatively under-specified.

Another solution is to provide a client in the form of an *applet.* An applet is a small Java program that is embedded into a web page. The program runs on the client side. Applets are programmed by extending the `applet` class and implementing the methods that are required to be overridden. The non-java equivalent are ActiveX controls, which are pieces of code that can be run using a web browser. Having an applet as a solution provides more functionality than say a web browser, but less, if no security requests are made, than an application. The applet needs to gain security rights from the user to do many things, including writing and reading to disc. Some users will have security issues against letting applets do this. Applets require the correct Java plug-in to run. Users may need to download and install this. Applets put greater demands on the resources of the host machine but this moves the processing away from the server.

Using a web browser as the client has many advantages. There are no installation tasks to perform (other than installing a working web browser, if required). The demands on the client are less than an application and the client can run on any platform that supports a web browser. The solution does put more

load on the server, but considering only a few people at any one time will be submitting MAP jobs, this should not cause a performance or load problem. There are problems with uploading large files using web browsers; HTTP was not designed to provide for the upload of files in the order of hundreds of megabytes.

8.3.10 Servlets

In the DUCK system the web browser acts as the client. HTML web pages are produced on the server and are displayed by the client. Servlet (see 5.5.3) technology was used to provided this user interface.

The servlets produced (described in 8.4.1) were written against the *servlet API version 2.2.*

Originally all the servlets inherited directly from `javax.servlet.HttpServlet`. The software was revised so that a base class `duck.web.Ducklet` was produced. Most servlets in the DUCK system inherited from this class.

8.4 Completed Software Overview

8.4.1 Packages Structure

The software was arranged into packages, as discussed in 3.3.4. At the start of the development process only one package was defined, as development proceeded the number of packages grew.

This section describes the different classes that build up the DUCK *web application* in detail.

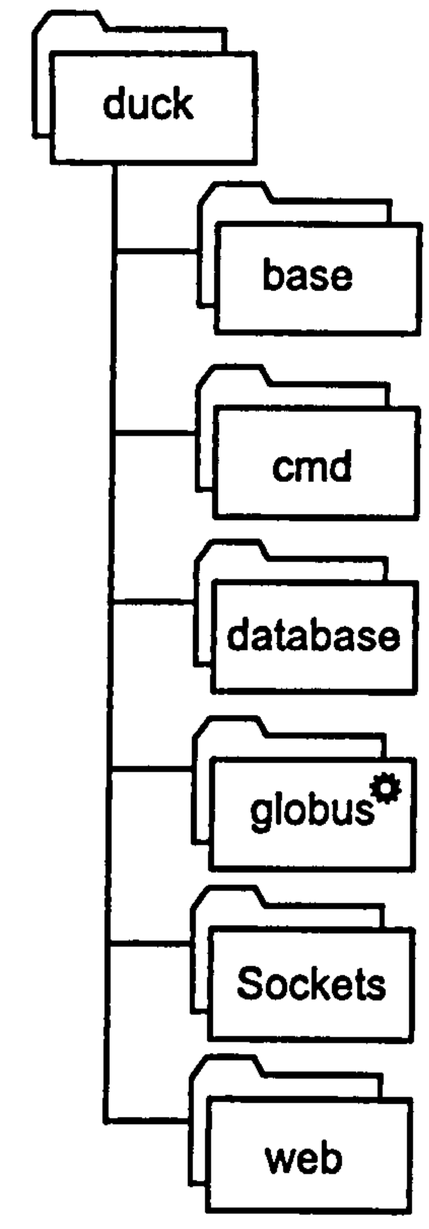

Figure 8.1: The package structure for the DUCK web application. * The package labelled 'globus' was an experimental package I produced and is not discussed in this section.

duck

The duck package. These are basic classes used many times in the project.

DUCKProperties

The class `DUCKProperties` is used to load parameters from a text file and make them available to the web application. The parameters or properties are divided into three categories;-

- Database related properties
- MAP Query system properties
- MAP Queue system properties.

getXXX() style methods make the properties available to the web application via the class `DUCKProperties`. The properties themselves are stored in a simple text file, which is read by servlets in the web application when the servlets initialise. The file is located in the folder pointed to by the Java system property `'user.home'`. This folder tends to be within the web server, although its location can vary a little between operating systems. Hence sometimes the file needs to be copied to the location specified in the exception, which is thrown when the DUCK system cannot find the properties file.

The properties file can be edited on the server with a simple text editor. Comments can be added to the file using '//' at the beginning of lines. The order of the properties in the file is important and should be maintained. Any changes to the file are not read until the DUCK web application is restarted. The database related properties include:-

- The *database driver* used to connect to the database with. This is the fully qualified Java class name of the JDBC driver. For example, this could be `'org.gjt.mm.mysql.Driver'`. This information is returned as a string by the method *getDBDriver()*.

- The *database location*. The location of the database, for example `'jdbc:mysql://localhost/map'`. With the MySQL database the user has to specify a database on the host. This information is returned as a string by the method *getDBLocation()*.

- The database *username* used to connect to the database. This should be something reasonably generic like `'tomcatuser'`. This information is returned as a string by the method *getDBUser()*.

- The `password` used to connect to the database with. Usual rules for choosing a secure password apply. The password is stored as plain text in the properties file, but the file is not publicly accessible on the web.

MAP Queue host properties include;-

- The queue host, the location of the MAP Queue as a machine name with network domain. For example `pc178001.ph.liv.ac.uk`. The information is returned as a string by the method *getMAPQHost()*.

- The number of the port to connect to on the queue host. This is the port number on the queue host machine to connect to. It is returned as an integer by the method *getMAPQPort()*.

MAP Query host properties include;-

- The query host, this is the location of the MAP Query host. This information, a machine name and network domain, is returned as a string by the method *getMAPQueryHost()*. For the current core MAP software it is usually the same host as the queue host.

- The number of the port to connect to on the query host. This is an integer returned by the method *getMAPQueryPost()*. The port for the queries must be different to the queuing port.

DuckLib

The `duck.DuckLib` class contains static methods and properties used throughout the DUCK project. The field EOL (end of line) contains the end of line terminator for the platform the program is running on. This obviously changes between operating systems and is obtained by the call `System.getProperty("line.separator")`. The method *makeDateStampe()* provides a standardised time and date stamp. An alternative to this solution was to use the java.text.DateFormat class. The method produces time/date stamps in the form of `YYYY-MM-DD_HH:MM.SSTT`, for example; `2003-12-05_17:16.49PM`. These stamps are easily sortable and are stored directly in the database system. The method makes a date stamp for the current time.

INetAddrParser

`duck.INetAddrParser` is a class that offers supplementary functionality to the class `java.net.InetAddress`. This extra functionality allows additional information about the address to be obtained, for example the network class and the domain. The class was designed to work with the existing model of MAP security, which checks the location of the connecting client. The class has 3 constructors;-

- The first constructor accepts a `java.net.InetAddress` as a parameter.
- The second constructor accepts a hostname as string.
- The final constructor accepts the numerical IP address as a string.

A private method *parse()* tokenises a string containing the numerical IP (`xxx.xxx.xxx.xxx`) into the 4 octets. This method is called by all three constructors. The following methods return the octets as integers;-

- *getOctet0()*
- *getOctet1()*
- *getOctet2()*
- *getOctet3()*

The method *getNetworkClass()* gets the class of network the IP address is located in, returning it as a human readable string. Possible values are 'A','B','C','D','E' and 'local loopback'. The following methods return a boolean value indicating whether the IP address is located in the following network classes;-

- *isClassLocalLoopBack()*, returns true if the IP address represents a local loopback, meaning octet zero has the value 127.
- *isClassANetwork()*, returns true if the the IP address is in a class A network, meaning octet zero has a value between 1 and 126.
- *isClassBNetwork()*, returns true if the the IP address is in a class B network, meaning octet one has a value between 128 and 191.

- *isClassCNetwork()*, returns true if the the IP address is in a class C network, meaning octet two has a value between 192 and 223.
- *isClassDNetwork()*, returns true if the the IP address is in a class D network, meaning octet three has a value between 224 and 239.
- *isClassENetwork()*, returns true if the the IP address is in a class E network, meaning octet four has a value between 240 and 255.

A static method *isAddressInDomain()* is provided in the class to test whether two addresses are in the same domain. The method *getWebServerPath()* returns the real path to a web server. The port to connect on is needed as an argument. The method *getHostName()* returns the resolved hostname of the host.

JobControlImpl

`duck.JobControlImpl` is an implementation of the `JobControl` interface. The class implements all the get and set methods defined in the `JobControl` interface. Parameters are stored internally as immutable strings. The JobControlImpl can be obtained as a JobControl interface by the method *getState()*. This means that the JobControlImpl can be used (with the stored values) in situations that require the use of the interface JobContol.

The `JobControlImpl` provides some methods that can perform basic checks on the data.

MAPJobStateImpl

The class `duck.MAPJobStateImpl` implements the interface `duck.base.MAPJobState`. The class implements all the methods that the interface requires and is used for storing information relating to a single MAP job. It holds the following six fields internally, the account, the user, the job status, the site, the estimated start time and the job name.

The class provides *getXXX()* and *setXXX()* methods to access the fields and a static method to get a blank instance of the class. There is a static method to convert a `JobStatus` type into a human readable string. The *toString()* method is overridden to return a string that describes the state of an instance of `MAPJobState`.

MAPQueryTask

`MAPQueryTask` is responsible for running a task that updates the locally stored states of MAP jobs and the job queue they are in. This state is stored in the MySQL database. The class uses the singleton model, so there is only ever one instance of MAPQueryTask. There is a static method for creating the instance of MAPQueryTask and another 'clean up' method for disposing of the timer instance.

`MAPQueryTask` has an inner class, a timer task, which has the task of updating the database by querying MAP periodically. The interval is constant and retrieval does not correspond to when users request the information from DUCK. The reason for this being that many user requests for the job queue would not lead the core MAP listener being saturated with requests.

Using `QuerySocket` the state of the MAP job queue is obtained and written to one of two database tables. There are two database tables, so that whilst one is being updated the other is available to provide query information to clients. After the updates on the inactive table are complete it is switched to be the active table. This means queries are answered with data from the recently updated table.

SingleResponse

The class `duck.SingleResponse` has two uses; to encapsulate *error*,*information* and *warning* responses from MAP and to encapsulate *name*, *value* and *description* triplets. The values for name, value and description are set when the object is constructed, these values are final (immutable) for the lifetime of the

object.

The class has 3 constructors, two of which I depreciated because I wanted the third to be the preferred choice. Differing amounts of information (the constructors arguments) are passed to the constructors. The preferred versions requires all the fields in the constructor.

Four methods return the values of the fields in the object, these are;-

- *getName()* returns the value of the name field as a string
- *getMessage()* returns the value of the message field as a string
- *getDescription()* returns the value of the description field as a string
- *isError()* returns a value indicating whether the object represents a value

There are no corresponding *setXXX()* methods, because fields in the class are immutable and unchangeable. Overriding of the method *toString()* provides a human readable dump of the object as a string.

SingletonException

This class is an extremely simple class, used to provide an implementation of the *singleton model.* Certain situations encountered in the programming of the DUCK web application require that there is only ever one instance of a particular class. This is the singleton model. A `SingletonException` is thrown when the creation of a second instance of a singleton class is attempted. The `SingletonException` was created by extending `java.lang.RuntimeException`. The resulting class has two constructors with differing parameters, the second parameter requiring a message of explanation, is the preferred.

duck.base

This sub-package defines interfaces that describe a few data structures on MAP in an object orientated fashion.

Account

The interface `duck.base.Account` is an interface that models an account in the MAP sense. On the MAP cluster an account is a grouping of users. Accounts have a limit on the amount of disk space that they may use. Also associated with accounts are sites. These are a list of sites that can submit jobs using the account. Implementing classes are required to use `java.util.Vector`'s to return some information fields, which have multiple values.

- *getAccountName()*, implementors of the interface should return the name of the account as a string when this method is called.
- *getSites()*, implementors of the interface should return the list of value sites as strings, encapsulated in a Vector.
- *getDiskLimit()*, implementors should return the current disk limit on MAP for the account in gigabytes, as a double precision number.
- *getUsers()*, implementors should return the list of users (usernames) that are valid users for the MAP account, as strings encapsulated by a vector.

JobControl

The `JobControl` interface is described in section 8.3.6.

MAPJobState

The `duck.base.MAPJobState` interface defines the information available about a single MAP job that has previously been submitted to MAP. Type-safe constants are used to classify the different job states. In Java a type-safe constant can be created by creating an empty, static, final class. Final instances of the class can be created to represent the various constants that are needed. The constants defined in the interface are as follows;-

- `QUEUEING`, the job has been queued on the MAP system.

- `WAITING`, the job is waiting on the MAP system.
- `LOADING`, the job being loaded on the MAP cluster.
- `RUNNING`, the job is running on the MAP cluster.
- `FINISHED`, the job has finished running on the MAP cluster.
- `UNDEFINED`, does not describe a job state, but is used to catch errors at runtime.

Methods in the interface define the information about a single MAP job. Corresponding get and set methods allow the information to be sent and obtained. These methods are listed below: -

- get and set JobName refers to the job name of `MAPJobState`.
- get and set User refers to the user the job belongs to.
- get and set Account refers to the account that the job belongs to.
- get and set Site refers to the site the job belongs to.
- get and set Status refers to the status of the job, this is described by type-safe constants discussed above.
- get and set EstStatTime refers to an estimated start time for the MAP job. This estimated start time is calculated by MAP and depends on the run times of the proceeding jobs.

duck.web

The front end to the DUCK application is provided by servlets, as discussed in section 5.5.3. Figure 8.2 illustrates the relationships between classes in the `duck.web` package.

All servlets are derived from the `HttpServlet` class, which is provided in the `javax.servlet` package. This `HttpServlet` is in turn derived from `GenericServlet`. `HttpServlet` provides methods for writing HTTP-based servlets.

Many of the servlets are subclassed from the servlet `duck.web.Ducklet`. This servlet provides functionality for checking whether the request is valid for each transaction. `Ducklet` has methods to obtain the session cookie (see 8.3.4) from the HTTP request and check it against the database system.

Many classes in the package are derived from Ducklet, taking advantage of its functionality.

ConfirmServlet

The `ConfirmServlet` is a servlet allowing the user to confirm they want to submit their job. By extending the servlet `Ducklet` the resulting servlet has the various functionality provided for it, for example, by sub-classes calling `Ducklets` *init()* method, various parameters are loaded on initialisation of the servlet. On each client call to the servlet, the base class `Ducklet` can check whether the request is valid and get the session ID of the client.

`ConfirmServlet` is called from the page produced by the servlet `SubmitJobServlet`. The user has two potential options; firstly, to run the job and secondly to cancel the job and delete it from the core MAP system.

These two options are presented as links to the user. Each link has encoded, as a URL parameter, a parameter confirm. The parameter *true* runs the job and *false* means the user does not want to run the job.

Once that the `ConfirmServlet` has retrieved the parameter it then needs to communicate it to the core MAP system. The servlet uses an instance of `JCBConfirmSocket` do this. The session ID is passed to link the request to the Job Control Block previously sent to the core system. After the information is exchanged the user then receives a confirmation page that informs him/her of the action taken.

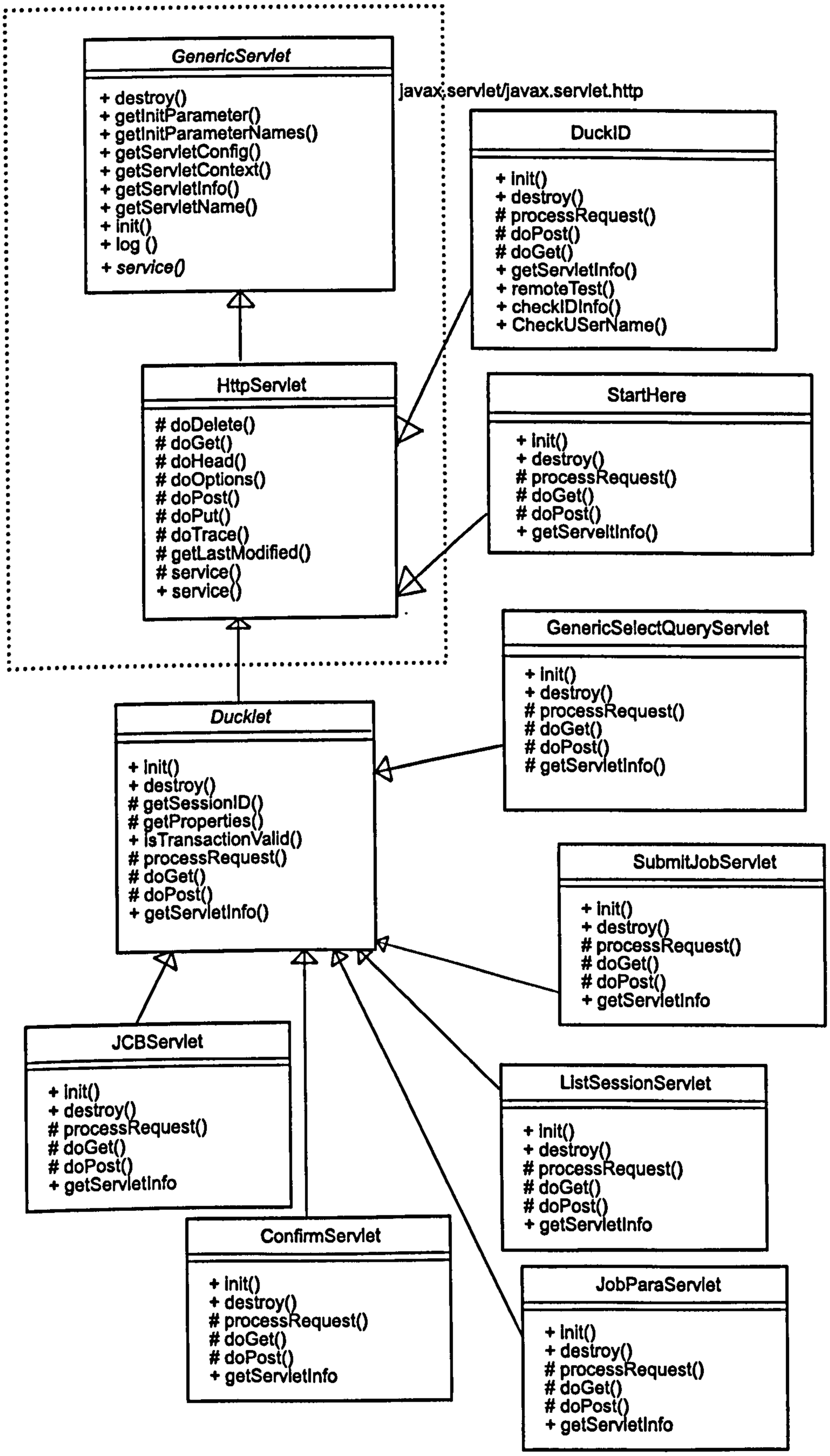

Figure 8.2: Inheritance relationships between classes in the package duck.web.

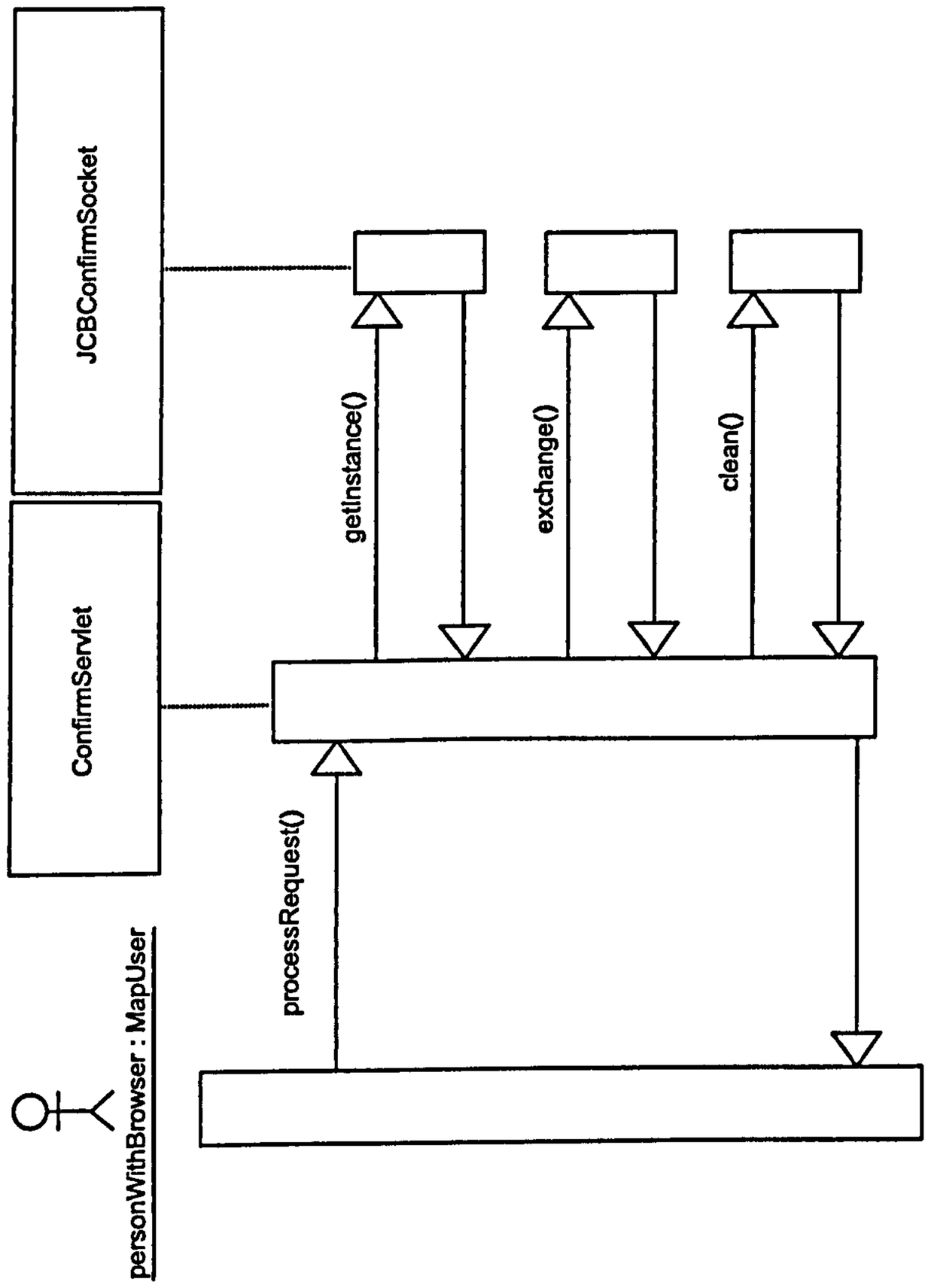

Figure 8.3: Confirm Servlet interaction diagram.

CookieTest

This servlet can be used to determine whether cookies are enabled in the user's browser.

DuckID

The servlet `duck.web.DuckID` is a servlet for controlling logging into the web application. The servlet does not extend `Ducklet` but instead extends `HttpServlet`. This is because the mechanisms to process users that have already logged in are not needed. Instead `DuckID` reads parameters from a web

page that is submitted to it to determine who is trying to log in. `DuckID` reads the clients IP address, this is used to determine the location of the user. The user volunteers his/her username and the account they are a member of, using the web pages form. If these two details are null (meaning no parameters have been passed from the webpage), the `DuckID` servlet has been called without a web pages submit being used. The servlet will then not process the request. After trimming the username and account, basics checks are made, such as whether they are blank or excessively long. The servlet then checks whether the user and account are valid by using the database (via the `AccountDB` class).

Once this has been done a session ID is generated. It is stored in the database and set as a cookie. When all of the above has been completed successfully a page is returned to the user giving them options of what to do next. The options are in the form of navigateable URLs, which invoke other servlets.

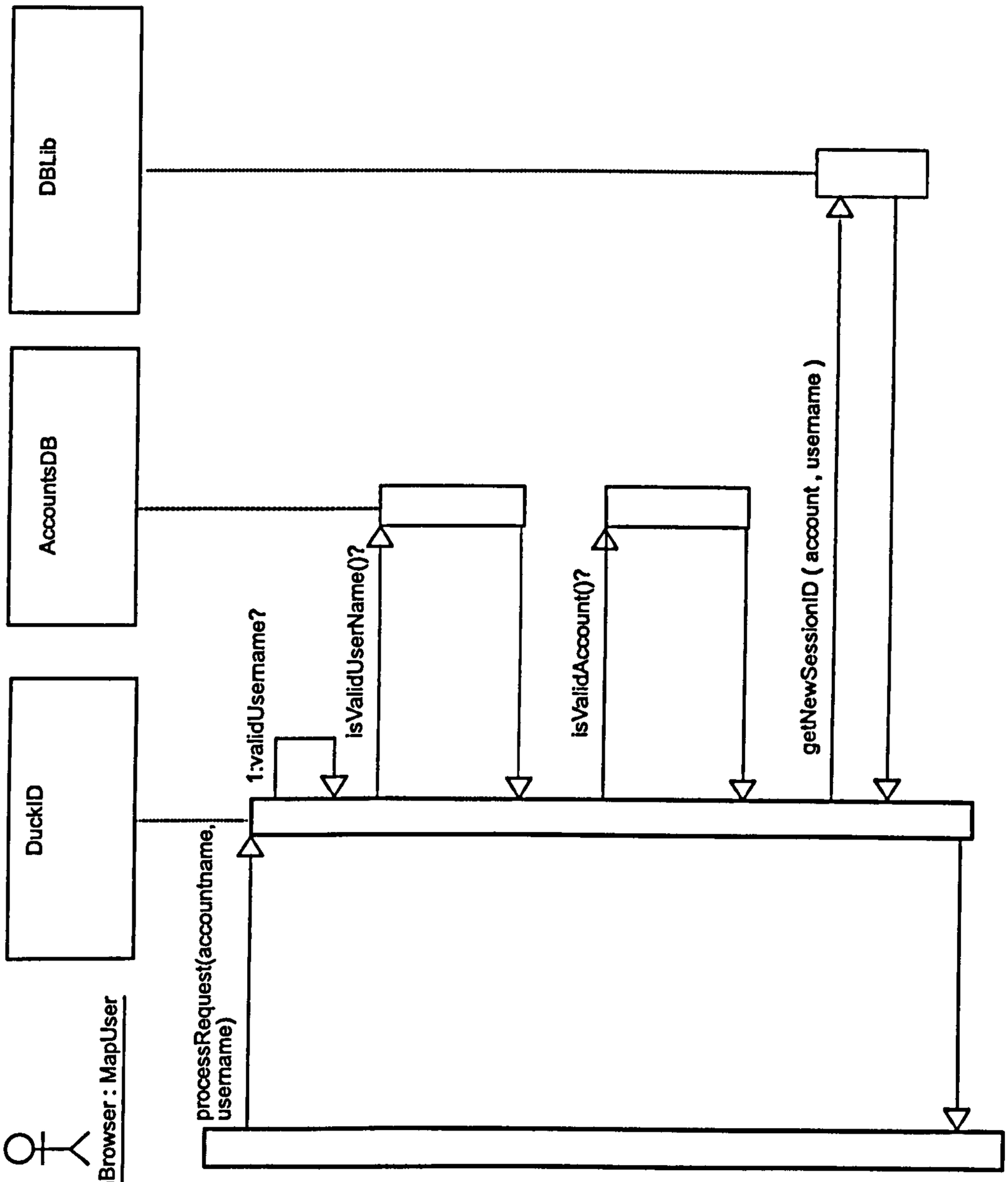

Figure 8.4: Interaction diagram for DuckID servlet.

Ducklet

The class `duck.web.Ducklet` is a servlet that acts as a base for many of the servlets in the DUCK web application. `Ducklet` extends the class `javax.servlet.http.HttpServlet`, which is an abstract class designed to be subclassed to create servlets that deliver HTTP.

The `Ducklet` servlet itself is abstract, only because there is no reason why an instance of the class should be created. Creating an instance would just authenticate the user for the request and then do nothing else. It provides some basic functionality required by the majority of servlets that form the DUCK web application.

Firstly, the settings required for the DUCK system are loaded by this servlet when it is initialised by the *servlet container*. This typically happens once in the lifetime of the servlet. The code for this is in the *init()* method in the servlet. The servlet will throw a `ServletException` if it cannot correctly load the settings.

Secondly, methods in `Ducklet` process requests from the client. The cookies sent by the client are obtained and processed, most important is the session ID cookie. A look-up in the database using this session ID is done, providing the system with the details associated with the session ID.

Thirdly, the servlet checks whether the user should use the system, this is done on each request made by the client. `Ducklet` checks whether the transaction is valid using the method *isTransactionValid()*. If the user's access to the system is valid (they have supplied a session ID that exists in the database) then *true* is returned, if there has been a problem logging in then *false* is returned by *isTransactionValid()*. Access to the *isTransactionValid()* method is *protected*, meaning that only classes that subclass `Ducklet` can use this method.

Fourthly is the availability of session ID. If a valid session ID is passed to the servlet and this session ID exists in the database then `Ducklet` makes the ID available via the *getSessionID()* method. This method returns the session ID for the user. Access to the session ID is protected, meaning only subclasses of `Ducklet` can access the session ID this way.

Finally, the method *getProperties()* makes the properties the DUCK system needs available to servlets that subclass this servlet.

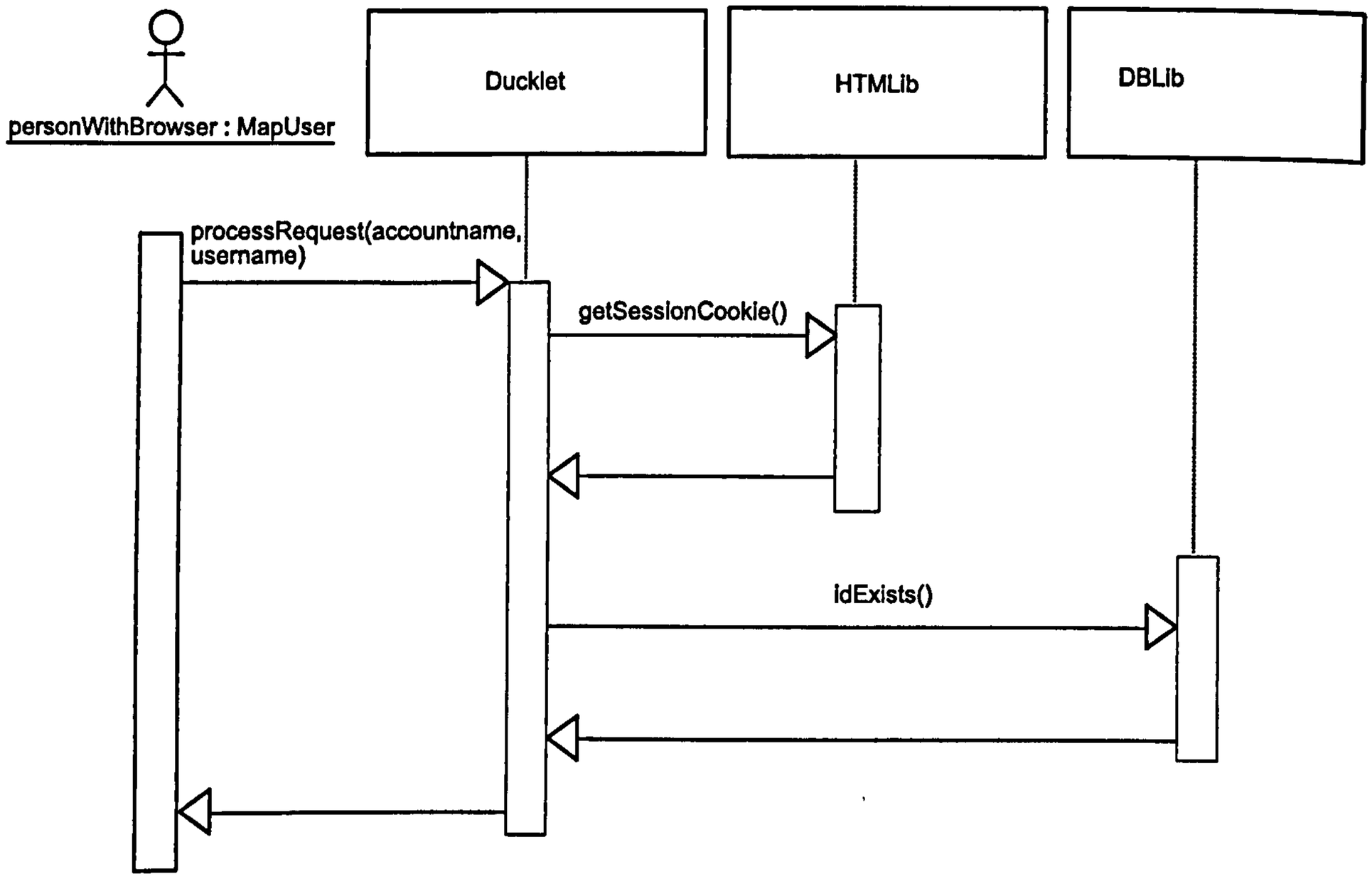

Figure 8.5: Interaction diagram for the abstract Ducklet servlet. A non-expert MAP user is the actor. Classes derived from Ducklet perform cookie fetching and ID checking, using the logic contained in Ducklet.

GenericSelectQuery

The `GenericSelectQuery` servlet is a servlet that allows sets of SQL queries to be run against the DUCK database. The queries are defined in a XML file which can be edited to add, enable/disable or remove the queries. Only SELECT queries are supported, these are queries that are read-only to the database system. The servlet, initially called without any parameters, lists all the available queries to the user, allowing the user to select a query to run. The chosen query can be selected by following a link which causes the servlet to call itself again with a parameter identifying the query to execute. The query is then executed and the results return in a table contained in an HTML page. The servlet is a subclass of `Ducklet`, reusing the session management provided.

The servlet uses an XML file on the server called `sql.xml`, a file with `queries`

as its root element. The queries element can have any number of query elements as children. Each query element represents a set of SQL queries. The query element can have a description, an XML attribute indicating whether the queries should be made available (publish) and an XML attribute indicating which account the queries are associated with. These pieces of information are stored as XML attributes in the query element.

Nested inside the query element are any number of SQL elements. These contain SQL queries, which are executed in the order they are listed. When the Servlet is initialised the SQL queries file is loaded. This means if the file is changed the web application must be reloaded for changes to take effect. For example added SQL queries will not be available until the web application is restarted.

The file is loaded using the method *getResourceAsStream()* which is in the servlets `ServletContext`. If the resulting `InputStream` is null, indicating a problem, then an `Exception` is thrown. Otherwise the `InputStream` is parsed into a DOM(see section 4.6.5). Once this has completed the data from the queries file exists in memory as a DOM that can be transversed to obtain the data in it.

Data is obtained from the DOM by calling the methods in the interface `org.w3c.dom.Document`. These methods allow navigation and retrieval of the data in the DOM. The method *queryNodes()* returns a `NodeList` which contains the nodes that contain the sets of SQL queries.

Internally the class uses a nested inner class, `QueryNodeData`, which encapsulates the data from a query node. This class has the following fields;-

- *name*, a string giving the query a name
- *description*, a string describing the query, which should be human readable.
- *publish*, a string indicating whether the query should be made available to users

- *account*, a string indicating which account the query belongs to
- *sql* an array of strings containing the SQL statements

The method *getNodeData()* takes a `Node` and extracts the data into an instance of `QueryNodeData`. This data is then in such a form that it can be easily digested by the methods that use this method. The method *listQueries()* lists the queries available. It outputs them as a web page with hyperlinks to allow each query to be run. The method *executeView()* processes a request to run a query set. It outputs the results of the query to a web page that is returned to the user.

HTMLLib

This class is an exception in the package duck; it is the only class that is not a servlet. Instead `HTMLLib` is a library of static methods for various common tasks. The methods are frequently called in the servlets that make up the DUCK web application. The methods that output HTML formatted output tend to require an instance of `java.io.PrintWriter` to which output is directly printed, instead of returning strings full of HTML. In the class there are methods for;

- Making error pages relating to handling exceptions. This method accepts, as its arguments, an exception instance, a `PrintWriter` and a string containing a human readable explanation. The exception can be any sub class of the class `Exception` or an Exception itself. A web page showing the error situation, stack trace and explanation is outputted to the `PrintWriter` instance.

- Making general error pages when an exception has not been thrown. This method outputs an error explanation web page to a `PrintWriter` when no exception has been thrown or in situations when it is prudent not to show the stack trace. A typical reason is not to divulge technical details of the system for security reasons.

- Getting the session ID from an array of cookies. This is a convenience method to get the session ID, as a string, from an array of cookies. If no cookies are returned from the web browser an exception is thrown explaining this.

- Making the HTML for a table and printing it to an open `PrintWriter` instance. The argument accepts table data as a `java.util.Vector`, which has more vectors representing the rows. Vectors containing the rows contain strings as their elements. The method loops through to get the rows and the cells contained in the rows and outputs the data marked up with HTML tags to the `PrintWriter` instance passed.

- Making a page that automatically redirects the browser to another page after a certain amount of time. The method produces a page with a message on it (passed in arguments), that allows the user to either wait 4 seconds to be redirected or offers a link the user can click on, to go to the page straight away. The automatic redirection is done using a meta tag http-equiv refresh tag.

- A method that makes the HTML for a displaying a `SingleResponse` object in a form. The method takes the `SingleResponse` object and obtains the 3 fields; name, message/value and description. All three of the resulting pieces information are written as a table row. The name and description are written to one table cell, with the name in bold. The message/value information is written another cell as an editable text input form control. As with other HTML methods the finished markup is written directly to the `PrintWriter`.

- A method that makes the start of an HTML form. Allows the method of submission to be set (POST/GET) and the action, which is the servlet the form is submitted to.

- A method that makes the end of a HTML form. The method finishes

the table off and adds a submit button that the user clicks to submit the form.

- Two methods to print out `Maps` and `Enumerations` as HTML formatted un-ordered lists. This allows data of these two types to be easily inserted into HTML output.

JCBServlet

The servlet `duck.web.JCBServlet` extends `Ducklet`. It provides the functionality to read submitted web pages containing the data the user has entered for a Job Control Block. See figure 8.10 for a screen shot of the web page and form used to enter this data. This data, once read by `JCBServlet`, can be used in the web application. Once the servlet has processed this data it then redirects the user's browser to the next stage of the process.

When `JCBServlet` handles a request the superclass methods in `Ducklet` are used to make sure the user's *session* is valid and obtain the *session ID*. If the session is valid the properties are read (see 8.4.2) and a blank Job Control Block object is created. This object is a class implementing the interface `JobControlBlock`. The parameters entered by the user on the web page are set as values in the Job Control Block.

It should be noted that Job Control Block parameters only relate to the setting up of the computing job, and not the simulation. The Job Control Block is cast to a class with the method *basicCheck()*, which is called to do a basic check. Using the class `JCBDBTalk` and the session ID, the Job Control Block is written into the database. A page is then returned to the user informing them the Job Control Block has been entered into the database. This page redirects to the next servlet after a pre-determined time.

JobParaServlet

`JobParaServlet` is a servlet that extends `Ducklet`. The resulting servlet allows the user to edit the parameters for their MAP job and then submit the values.

Once it has been checked that the user has a valid session, the *jobname* (the name of the saved job) they are running is required so that the correct default parameters for the job may be loaded. These parameters will relate to the physics or engineering job (for examples) that is being setup.

The *jobname* is obtained by getting the user's Job Control Block from the database (using the *session ID*). The *jobname* is found in the Job Control Block. Once the saved job name is known, a `SingleResponseDB` object is used to read the default job parameters back from the database. The parameters are read back as a name, a description and a default value for each parameter. This triplet of information is encapsulated in a `SingleResponse` object. Each `SingleResponse` is represented as HTML form fields on the web page sent back as a response to the request. The resulting page is shown in figure 8.11. The user can then edit the values for the parameters and submit the form.

Completed forms are submitted via the POST ([68]) method to the `SubmitJobServlet`.

ListSessionServlet

The `duck.web.ListSessionServlet` is a servlet responsible for providing listings, performing actions and showing the state of the MAP system to the user. Its function is determined by the *view parameter* passed to it when a request is made to the `ListSessionServlet` servlet.

The tasks performed by the servlet include:-

- showing the state of jobs in the MAP queue (for example the*jobname*, the *estimated time until a job completes* and the *state* of a job).
- setting all *session ID*s as inactive, meaning all current and past users

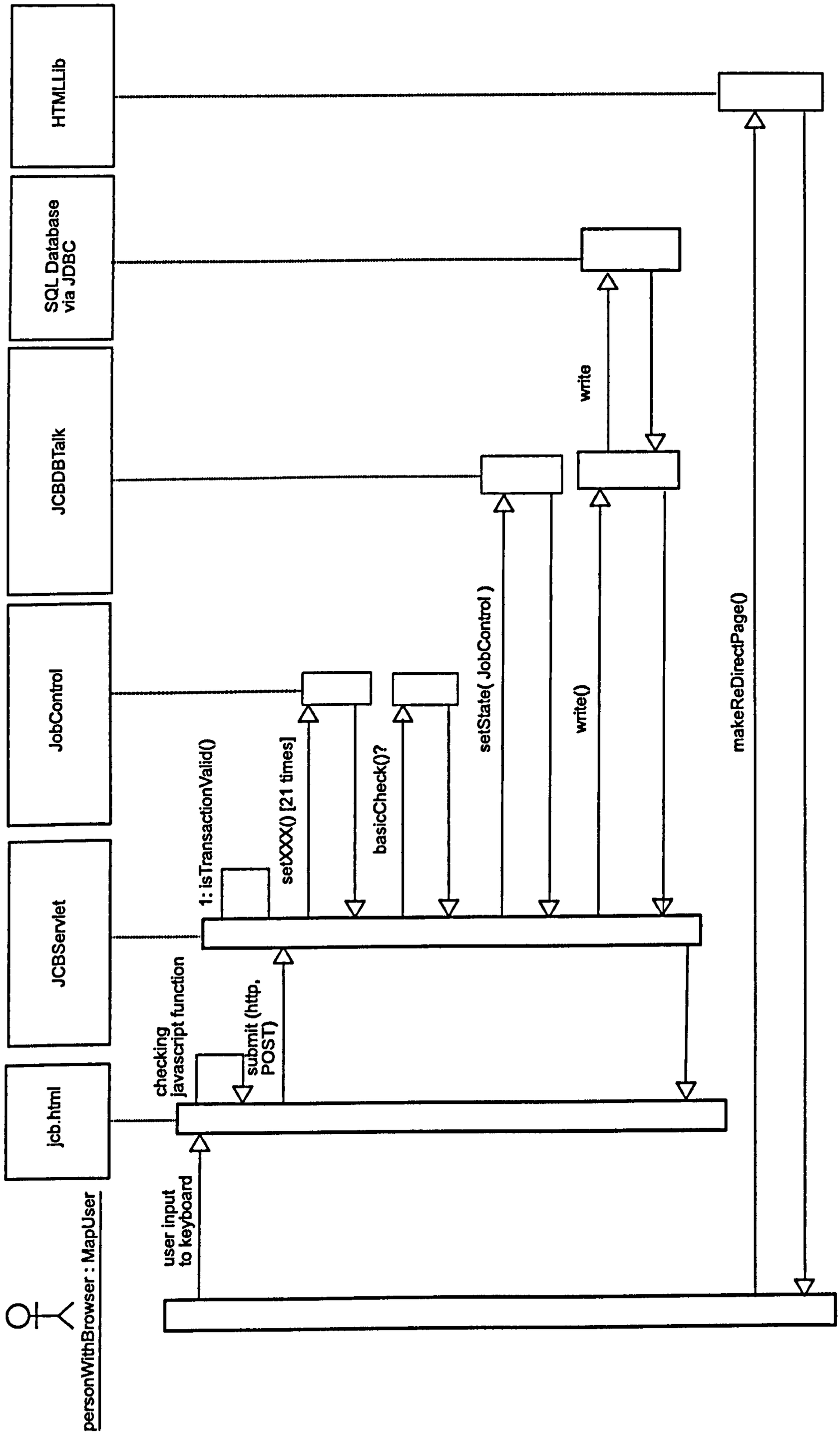

Figure 8.6: JCBServlet interaction diagram

are logged out.

- describing the schema of the DUCK database system.

`ListSessionServlet` starts the `MAPQueryTask` when it is initialised using the *init()* method. `MAPQueryTask` is a singleton class; meaning only one instance of it can be created by the `ListSessionServlet`. This is to stop multiple queries to the core MAP software being made. `MAPQueryTask` is a timer-based system that queries the core MAP system via a network socket.

If a request to see the status of the MAP queue is made, the information is obtained from the database, not directly from MAP. The database is updated on regular basis (120 seconds typically) from MAP using `MAPQueryTask`. This is to stop every user request generating a query to MAP. The information returned to the user is from one of two database tables updated alternately, so that one table is always available.

All the *session IDs* can be made inactive using `ListSessionServlet`. This could be used to log off all users when the DUCK web application needs to be restarted or shutdown. Sessions are made inactive by setting all the records in the active field as inactive in the database table.

The final function of the servlet is to show the database schema in tables on a web page.

StartHere

This is an extremely simple servlet written to redirect users to the login page.

SubmitJobServlet

The function of the servlet `duck.web.SubmitJobServlet` is to obtain the simulation parameters passed to it from the web page. The *session ID* and whether the session is valid for the transaction are provided by sub-classing `Ducklet`. Once a valid session ID is obtained the parameters are read from the web page

submitted to the servlet. Care is taken not to read the value of the submit button, as we do not need the value and it would add an extra, unwanted, value to the process. `SingleResponse` objects are created from the parameter name, value pairs. The related Job Control Block is obtained from the database using the session ID. If all the above processes are successful then the servlet tries to submit the job to the MAP cluster. This is done using an instance of `JCBSocket`, which is a class derived from `MAPSocket` designed to pass Job Control Block data and simulation data to a listener on MAP. The class is used in the following manner

- getting an instance of it, which points to the MAP queue daemon host
- setting the Job Control Block and simulation data for the instance
- Calling the *exchange()* method which opens communication, exchanges the information and then closes the connection.
- Obtaining the error/information/warning feedback from the instance.
- Cleaning up the instance of the `JCBSocket` because it is a singleton and the particular instance in question will not be used again.

After the information exchange to MAP, a web page is constructed with the feedback from MAP. As stated earlier this feedback is split into three areas, information, warnings and errors. The final action of the servlet is to provide links on the output page to allow the user to continue. If the user has errors they only have the option of cancelling the job. If the user has warnings, but no errors then they are given the choice of running or cancelling the job.

duck.database

The classes described below provide a 'binding' between the database and the DUCK application. They allow data to be stored persistently using the database. See section 4.7.2 for a description of relational database systems, a description of JDBC is given in section 4.7.4 and possible solutions to the problem of persistency are discussed in section 8.3.4.

AccountsDB

The class `duck.database.AccountsDB` stores and retrieves information about the MAP account system in the DUCK database. There is a reliance on `DBLib` for the underlying database access. The static methods provided by `AccountDB` allow various tasks to be performed without creating an instance of `AccountsDB`.

- *isValidAccount()*, takes an account name as a string and checks whether it is valid and whether the user is a member of the account. The result is returned as a boolean.

- *isValidUserName()*, takes a user name, as a string and checks whether it is a valid user. Again the result is a boolean.

The class implements, as an inner class, `duck.base.Account`. Instances of this inner class can be returned to provide information on individual accounts. The method *getAccount()* returns account information on an account specified by an account name passed to the method. A list of account names on the MAP system can be obtained by calling the method *getAccountNames()*, which returns the names of the accounts as strings contained in a vector. The number of accounts is available, as an integer, via the method *getNumAccounts()*. Calling the method *getData()* on an instance of `AccountsDB` populates the instance with the latest user, site and group data.

The inner class implementation of the `Account` interface, as described earlier, provides read only access to the data on an account. The following methods make information about individual accounts available;-

- *getAccountName()* gets the name of the account

- *getDiskLimit()* gets the disk space limit for the account

- *getSites()* gets a list of valid sites for the account

- *getUsers()* gets a list of valid users for the account

DBLib

The `duck.database.DBLib` is a class full of static methods that perform common database tasks in the DUCK project. Some methods perform very generic tasks, such as the method for getting a database connection object, other methods do slightly more specialised tasks; for example the method to check whether a session ID exists in the database.

The *getDatabaseConnection()* is a method that is frequently used throughout the web application to obtain a database connection. The method throws all exceptions that may occur rather than handling them as it was considered more appropriate to handle the exceptions elsewhere in the software. Specified in the properties file is which database driver to use and also the database host that is to be connected to. A `java.sql.Connection` object is returned which forms a connection to the SQL database defined in the properties file.

When using the method one must remember to close the connection object when it is finished with.

Using the method *getNewSessionID()* a new session ID for a user that is logging on can be obtained. A connection to the MAP database is made and a random number generated. This number is written into the session table in the database along with other information about the user and their current session. The same random number that is stored in the database is also returned by the method.

Using a sessions ID, the method *setSessionInactive()* makes a session inactive. This method is to allow a single user to log off or be logged off. The entry for the session in the session database table is flagged as active='false'. This means the user's session is now permanently inactive and they would need to log in again to use the system.

The method *setAllSessionsInactive()* works in a similar fashion to *setSessionInactive()* except it makes all currently active sessions inactive. All users would then be required to log in again. The method is of potential use when the entire

MAP system is being shutdown for maintenance.

getRandom() returns a random number (as a long precision integer). This number is used as a session ID and negative values can be returned. This method is used internally by *getNewSessionID()*.

The method *idExists()* is used to check whether a session ID exists and is active. The result of the check is returned as a boolean value. The method requires the session ID to be checked in a database table. Typically this is the session table. Also stored in this table is a field indicating whether the session ID is marked as active. If the session ID is not marked as active, then false is returned by this method. A false return value can also indicate that the session ID does not exist in the table. A true return value is returned if the session ID exists and is marked as active.

The method *idExistsSingleFieldCheck()* is similar to *idExists()*, but omits the check on the session ID being active. This means it can be used with database tables where the state of the session ID is not stored.

The method *getResultsAsVecs()* is a useful method for getting the results of SQL queries as a set of nested vectors. Using the method *getDatabaseConnection()* the SQL query passed as an argument is executed. The resultset is then packaged into a set of nested vectors [1].

If the resultset returned by the SQL query contains no data then a vector containing a row with the string "no data" is returned. There are methods in the `duck.web` package to turn the resulting vectors as HTML tables for display in web pages.

[1]The nested vector is structured as follows, for each row a vector is used to store each cell in the row as a string. The vector representing the row is stored in another vector that is equivalent to the whole resultset. The column names for each row are stored in a vector which inserted into the main vector before the first 'row' (vector) of data.

JCBDBTalk

The class `JCBDBTalk` is for storing and retrieving Job Control Blocks in the database system; meaning it proves persistency for Job Control Blocks. Instances of `JCBDBTalk` are constructed using a single constructor, which accepts the session ID which relates the Job Control Block to its session. This session ID is stored internally as a final field, meaning that the instance of `JCBDBTalk` is fixed to that particular session ID. The constructor also loads the properties needed to communicate with the database system.

`JCBDBTalk` works by having an instance of a Job Control Block stored internally. This can be obtained by using the method *getState()* or overwritten by using the method *setState()* which requires the new Job Control Block to be passed as an argument. The Job Control Block stored internally can be written to the database using the method *write()*, or can be over-written by obtaining a Job Control Block from the database using the method *read()*. Using the four methods above, Job Control Blocks can be stored and retrieved in the database for later use in the web application. Internally `JCBDBTalk` manages writes to the database with two private methods; *updateJCB()* and *insertJCB()*. If the Job Control Block has not previously existed in the database table *insertJCB()* is used to insert a Job Control Block with blank fields and the correct session id. *updateJCB()* is used to write the values from a Job Control Block into either blank fields in the database table or overwrite existing fields.

QueryDB

The `QueryDB` class is responsible for updating the MAP job status query information in the database tables that provide the query information to the users. This procedure is described in section 8.4.1

The static method *getCurrentTable()* returns which table should be accessed to provide information to clients. Another static method, *getTimeStamp()*, returns the time stamp for the requested table, thus allowing the web application to see which is the most up to date table.

There are two constructors for the class; one is a no-arguments constructor that simply allows an instance of `QueryDB` to be created, the other accepts a vector and a string. The vector should contain the data which describes the state of the MAP job queue and the string is the time stamp of the data. The constructor then updates the correct database table. The public instance method *update()* updates the correct table with the information supplied in the vector passed to the method. The method also requires, as an argument, the time stamp encapsulated as a string. Internally the class uses a private inner class, which has three fields, to help with processing the result sets. There are the following private methods in the class;-

- *clear()* clears the database table requested, which is identified using the integer passed in the arguments.

- *getTableName()* identifies the name of the database table from the integer passed as an argument. Typically table 1 is called q1 and table 2 is called q2.

- *populateTable()* populates a table with MAP queue data. The vector passed as a parameter has the data. The method also requires the number of the table to populate and an open database connection object.

- *setTable()* This method is used to set the states of the tables (1 or 2). The tables can be set to 'use'-boolean value true or 'not use' boolean value false.

- *getCurrentTableI()*. This method returns the number of the table that should be read from. It can return 0 if it cannot determine which table should be read from.

- *setTimeStamp()*. Sets the time stamp for a given table. The method requires the table number, the time stamp and an open database connection object.

- *testQueryInfoTable()*. This method is used to test the database table that holds the query information. The table should have 2 rows, one

referring table 1 and one referring to table 2. If there are more than 2 rows then this method is used to detect this potential problem.

SingleResponseDB

The class `duck.database.SingleResponseDB` reads and writes `SingleResponse` objects to the database system. Grouping of these objects make up the job parameters for computing jobs. This class writes a group of parameters into a single database table. Therefore, each separate set of job parameters has its own database table.

An instance of SingleResponseDB can only be used for the set of parameters it was constructed for; the jobname argument is stored as a final variable in the constructor. A database table is used to link jobnames to the tables that contain the parameters for the jobs. For operations that involve reading and writing sets of parameters in the database; a set of vectors stores the parameters internally. The *setState()* method can be used to set these parameters and the *getState()* method can be used to obtain the parameters (that have typically been read back from the database). The method *updateTable()* is used to write the parameters stored internally in the class to the database table for that job. The method overwrites the previous set of parameters with new parameters. Using the method *readTable()* parameters can be read back from the database and made available to the program. A number of methods provide information about the grouping of parameters;*getJobDescription()*, *getJobTableName()*,*getJobName()*, *getUser()* and *getGroup()*.

There are a number of static methods in the class, these are;- *createTable()*, *addOneRow()*, *deleteTable()*, *findTableName()* and *getAllJobs()*

duck.sockets

The classes that provide DUCK-MAP communication. Figure 8.7 illustrates the inheritance relationships in these classes. A base class `MAPSocket` provides code for the basics of socket communication and a framework for derived classes

(via abstract methods).

The derived classes, `JCBSocket`, `QuerySocket` and `JCBConfirmSocket`, extend `MAPSocket` implementing the abstract methods and adding methods where appropiate.

Typical usage of the derived classes follow the method call pattern of *open()*, *exchange()* (which does the work of exchanging data) and *close()*.

Some derived classes have a *getInstance()* method. This is usually so the class can follow the *singleton model* of only one instance at a time. See [69] for details of the singleton design pattern.

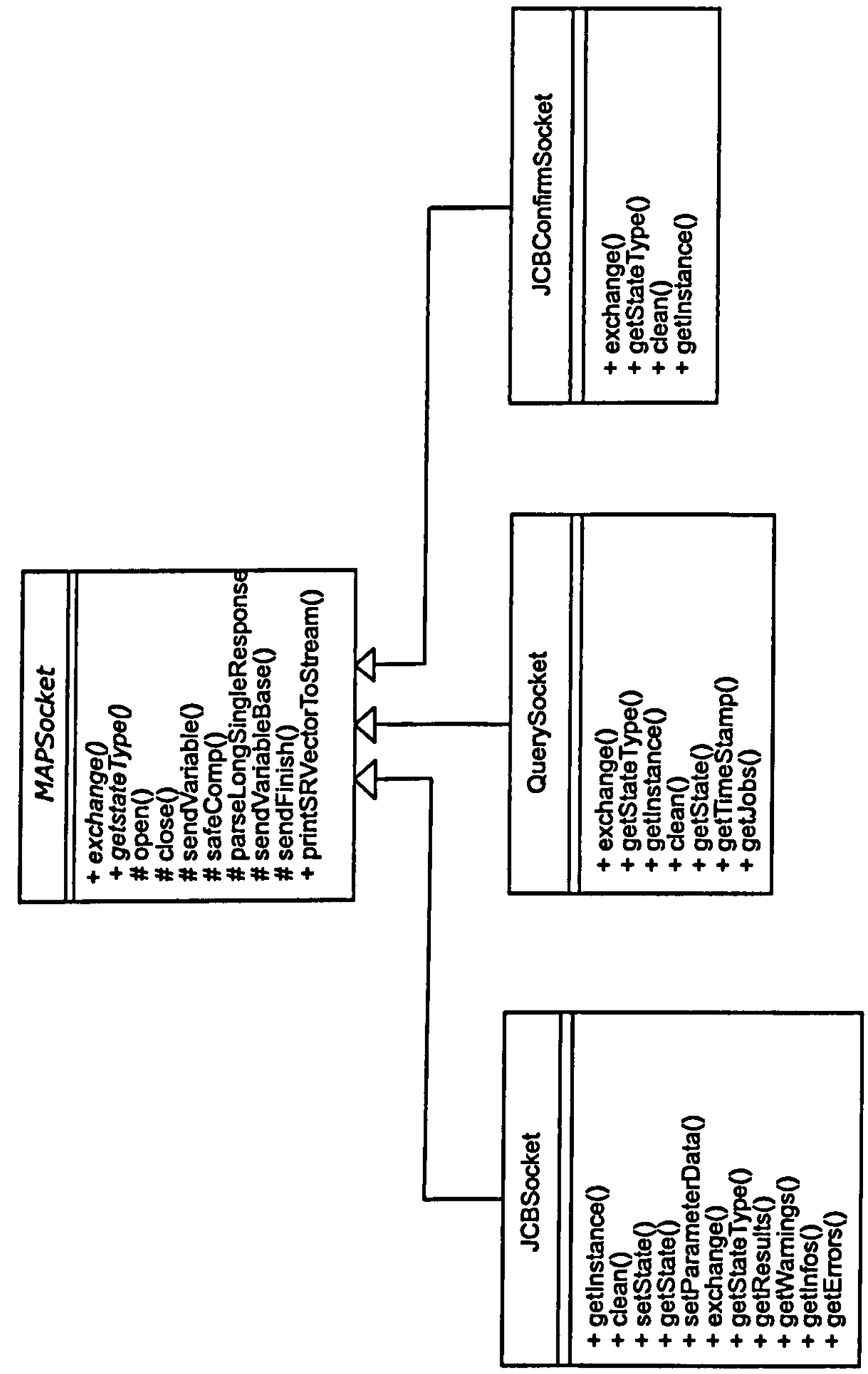

Figure 8.7: Relationship between classes in the package duck.sockets

JCBConfirmSocket

The class `duck.sockets.JCBConfirmSocket` is for confirming that a job should be run. When the details of a computing job have been submitted to the core tier of MAP, MAP returns a list giving messages in 3 categories, *errors*, *warnings* and *information*.

A job containing any errors cannot be run. A job that has warnings can

be run if the user decides they want to run the job. `JCBConfirmSocket` is for communicating the user's decision to MAP. The decision will be a simple `true` (run the job) or `false` (cancel and delete the job). The session ID is also passed, to identify which job we are talking about. `JCBConfirmSocket` extends `MAPSocket`, which provides a lot of the basic functionality. `JCBConfirmSocket` performs the following tasks;-

- Read configuration, to find out where the MAP Queue daemon is hosted, etc.
- Open socket, opens a socket to the MAP Queue daemon
- Send messages, sends the message (by overriding the *exchange()* method)
- Closes socket, closes the socket.

`JCBConfirmSocket` uses the *singleton model*, meaning only once instance of the class can exist at a time.

JCBSocket

The class `duck.sockets.JCBSocket` extends the base class MAPSocket, which provides a lot of the basic functionality. The class is designed to implement socket communication of Job Control Blocks and simulation parameters. `JCBSocket` performs the following tasks;-

- Read configuration, to find out where the MAP Queue daemon is hosted, etc.
- Open socket, opens a socket to the MAP Queue daemon
- Send: sends the Job Control Block (by overriding the *exchange()* method)
- Send: The class also sends the parameters relating the simulation.

- Receives the overall messages, which fall into three categories (Error, Warning and Information)

- Closes socket, closes the socket.

`JCBSocket` uses the *singleton model*, meaning only once instance of the class can exist at a time. The class stores an instance of Job Control Block internally. This can be retrieved using the *getState()* method or set using the *setState()* method. Simulation data is set using the method *setParameterData()*, which accepts a vector full of `SingleResponse` objects.

The class provides three methods; *getWarnings()*, *getErrors()* and *getInfos()*. These methods return the feedback information from MAP as vectors full of `SingleResponse` objects. The order of the first seven items is important when the Job Control Block is sent using the overridden *exchange()* method.

MAPSocket

The abstract class `duck.sockets.MAPSocket` acts a base class for the different classes that implement socket communication in the DUCK web application, see 8.7. The class was written late on into development of the software to stream-line the socket code and make it more maintainable. Classes that extend `MAPSocket` have access to a variety of protected methods in the base class, they also have to provide implementations for some abstract methods. `MAPSocket` encapsulates the basics of communicating between the DUCK web application and the core MAP cluster software. Communication is provided by network sockets; Java has the `java.net` package which provides implementations of various network software elements. A large part of the development process was spent designing a 'protocol' to allow the DUCK system to communicate with MAP. The principle of an exchange of data between MAP and the web application involves the follow: -

- The MAP software, on the host COMPASS node, has a listener on a particular port waiting for a communication.

- The web application initiates communication, by opening a socket to the listener.
- Variables are transmitted, with a possible response to each (a kind of question and answer protocol)
- There is a possibility for the sending of information by MAP to the web application after the variables have been communicated.
- The web application signals it has finished transmitting information and the socket is closed.

QuerySocket

The class `duck.sockets.QuerySocket` is designed to provide a means for DUCK to get information from MAP about the state of the MAP job queue. `QuerySocket` extends `MAPSocket`, which provides much of the base functionality.

duck.cmd

This package contains a single class `DuckConf`.

DuckConf

`duck.cmf.DuckConf` is a command line utility for configuring DUCK. It has limited functionality for adding users, adding user groups and editing saved jobs. The interface is a command line interface and the web application has to be stopped to use it. Typically the program would be run on the same machine as the web application, perhaps via *Secure SHell*(SSH).

8.4.2 Software Structure

Job Submission Queue interface

The Job Submission Interface consists of the following components;-

- `JobControl` interface which defines the Job Control Block
- `JobControlImpl`, an implementation of the Job Control Block
- `JCBServlet`, a servlet for reading a web page and getting the user's entered Job Control Block
- `JCBSocket`, a class for communicating the Job Control Block to the core MAP system.

Setup parameter loading

A recap of the parameters[2] needed from the system are; the database driver class name, the database location, the database username string, the database *connect* password, the MAP queue host, the MAP queue port, the MAP query host and the MAP query port.

The text based parameter system was discussed in section 8.4.1.

A new XML based system was devised and written, although there was not enough time to integrate it into the software. The underlying technology used, XML, is discussed in section 4.6.1.

The property file is loaded from the location specified by the Java property `user.home`. This can vary from operating system to operating system. When the system was deployed on Linux, an exception was generated because the properties file was not found in the location pointed to by `user.home`. The solution was to copy the properties file to the location expected (which was given by the exception message).

[2]A more detailed description of the parameters is provided in 8.4.1

Database System

MySQL is a relational database system. It was used to provide persistency for the DUCK Web-Application. MySQL provides the majority of the ANSI-SQL standard (see D).

The web applications was linked to the database using JDBC and the mm-MySQL driver. Classes under `duck.database` act as the database bindings and typically issue SQL statements, process the resultset and then provide the data in the form of Java variables and class structures.

The database schema consists of a number of tables;-

- `Session` for storing session data
- `Accounts` for defining accounts (groupings of users)
- `Users` for storing the users
- `Sites` for defining the valid sites that can have job submitted from them
- `Jcbs` for storing Job Control Blocks
- `parainfo` for storing the information about the query tables and which query table to use
- `q1` one of two tables for storing query information (MAP queue/job status)
- `q2` one of two tables for storing query information (MAP queue/job status)
- A table is created dynamically for each saved job defined. Each table is listed in parainfo.

The XML loading system for `SQL SELECT` queries. This aspect of the software was discussed in section 8.4.1, `duck.web.GenericSelectQuery`.

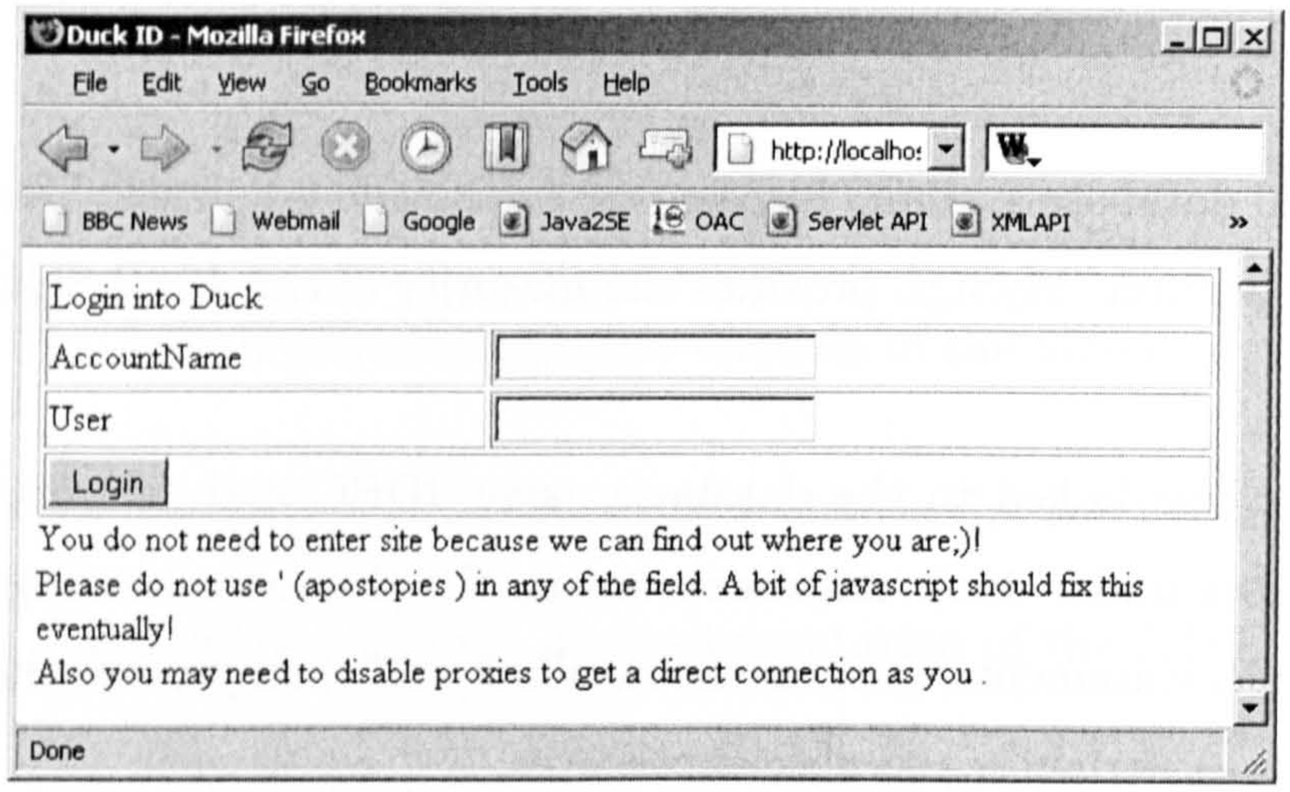

Figure 8.8: The web page for user login.

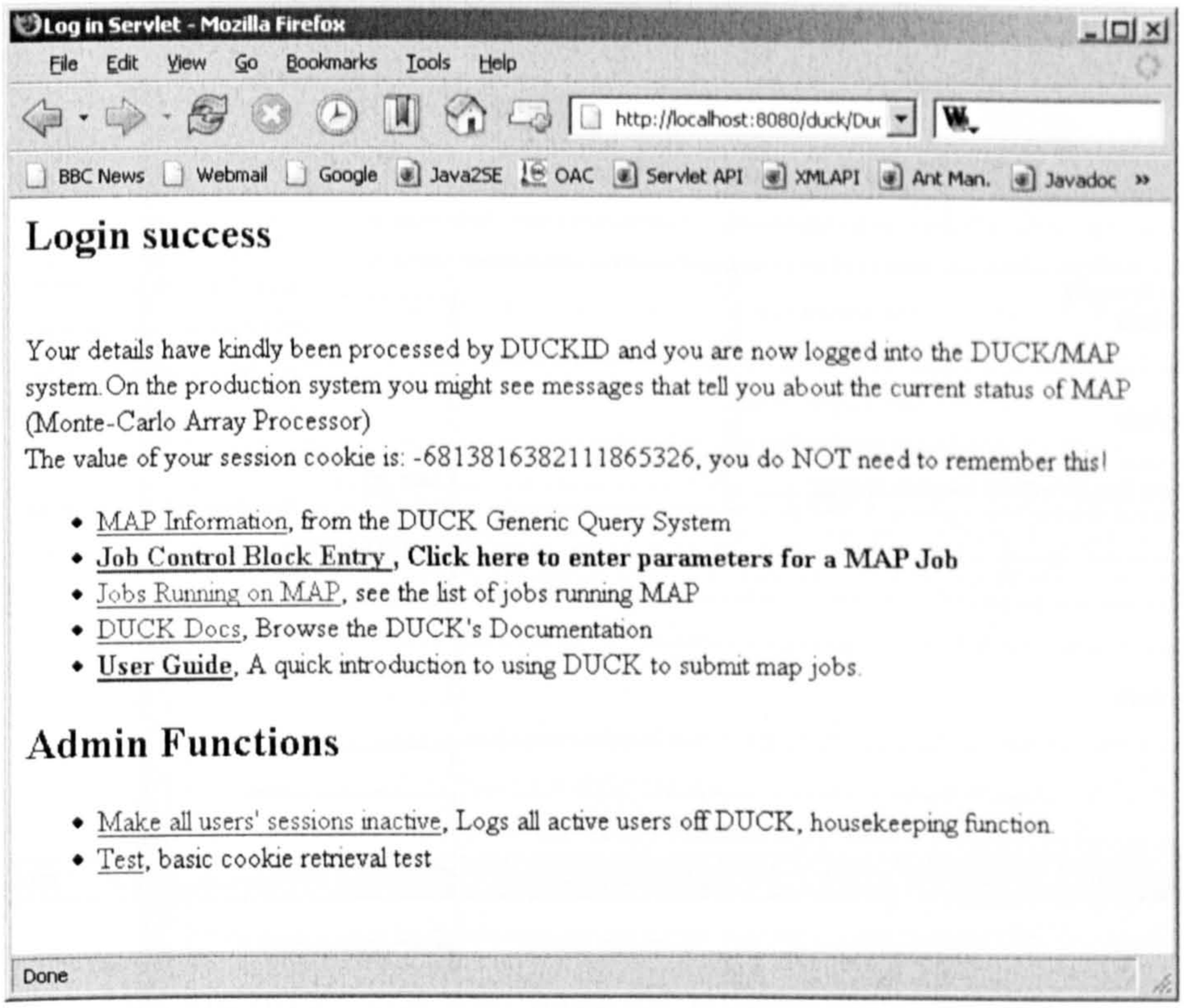

Figure 8.9: The web page displayed after the user has logged in. This page allows various actions.

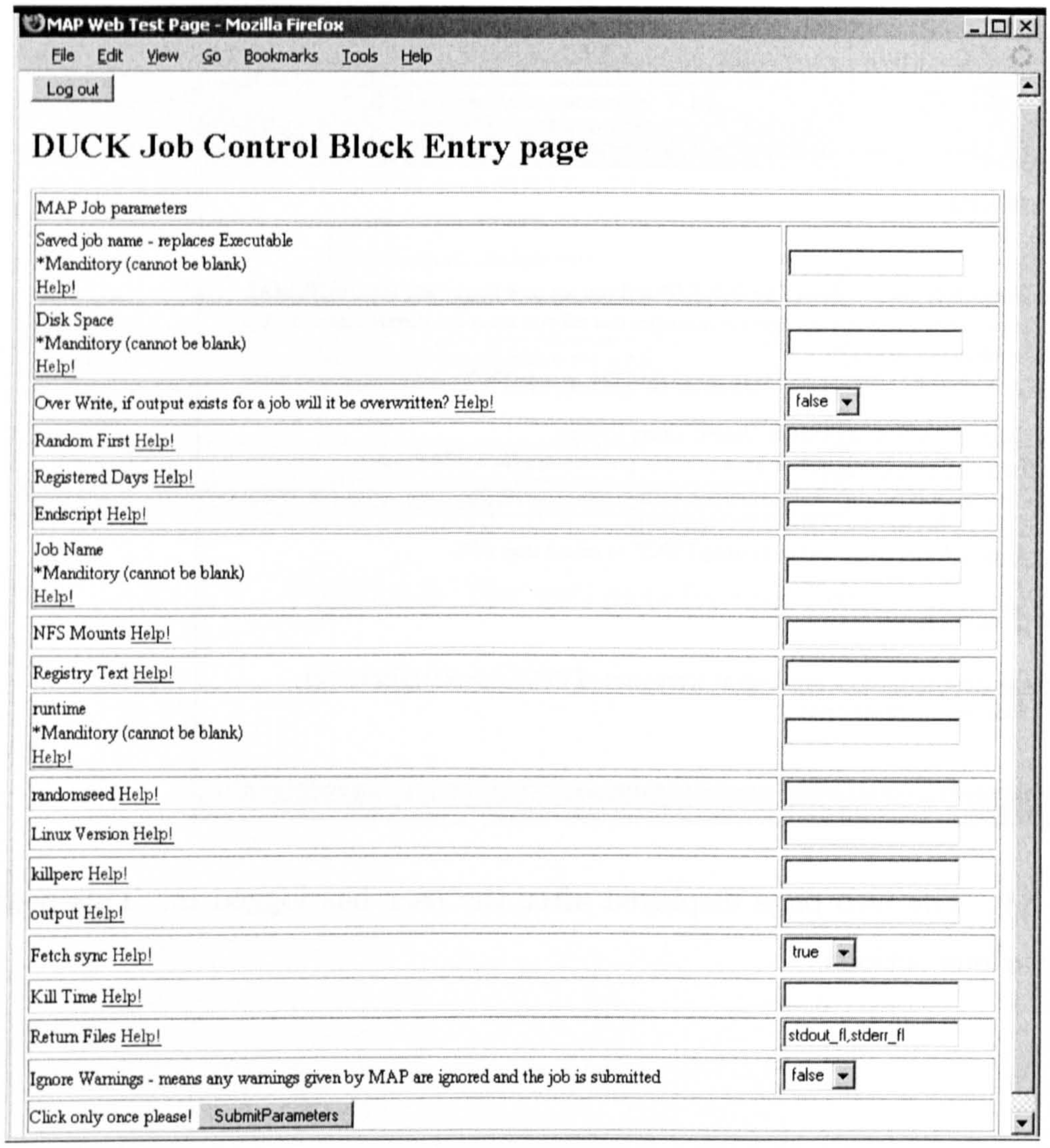

Figure 8.10: The web page that allows editing of Job Control Blocks.

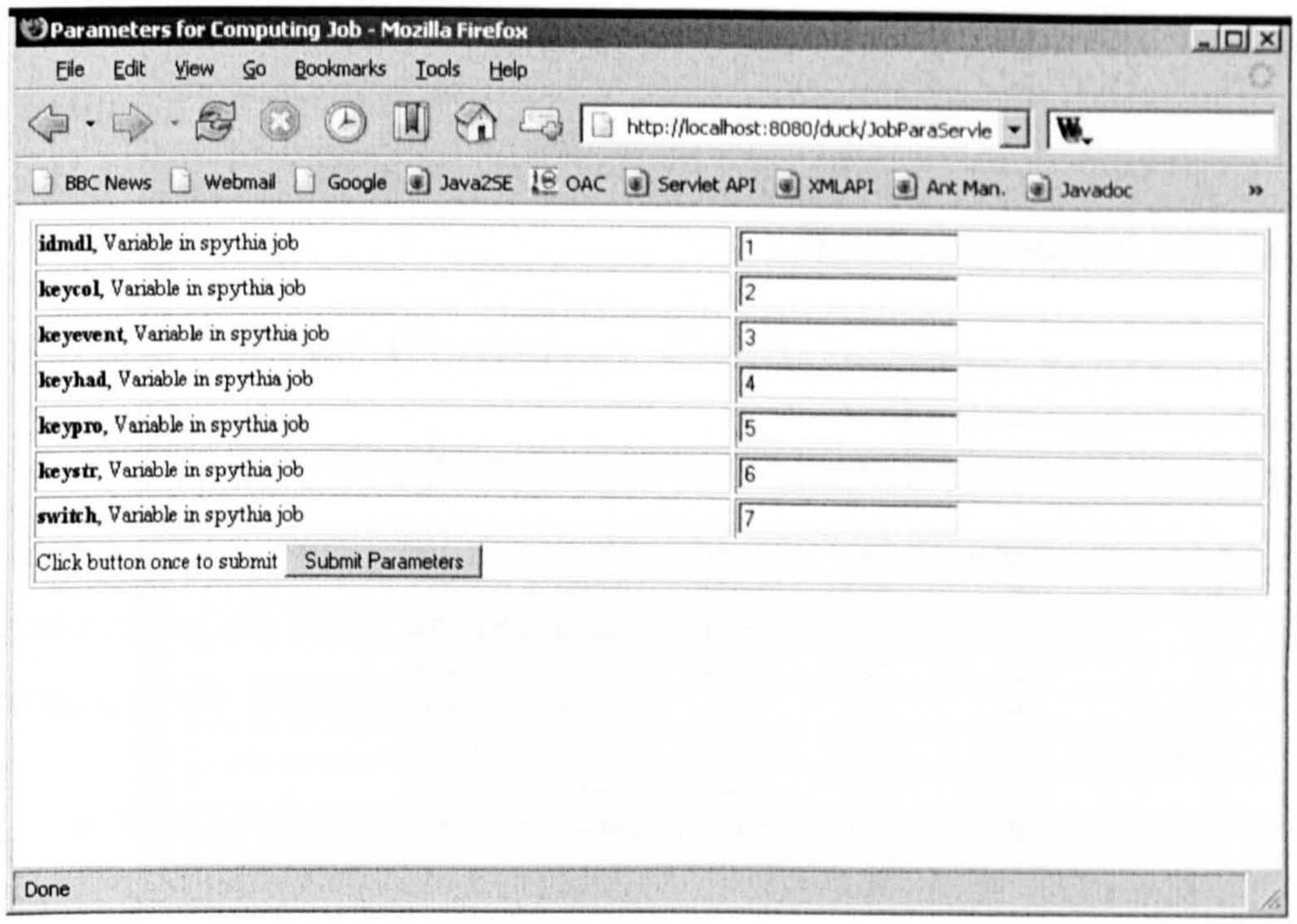

Figure 8.11: The web page for editing the simulation parameters

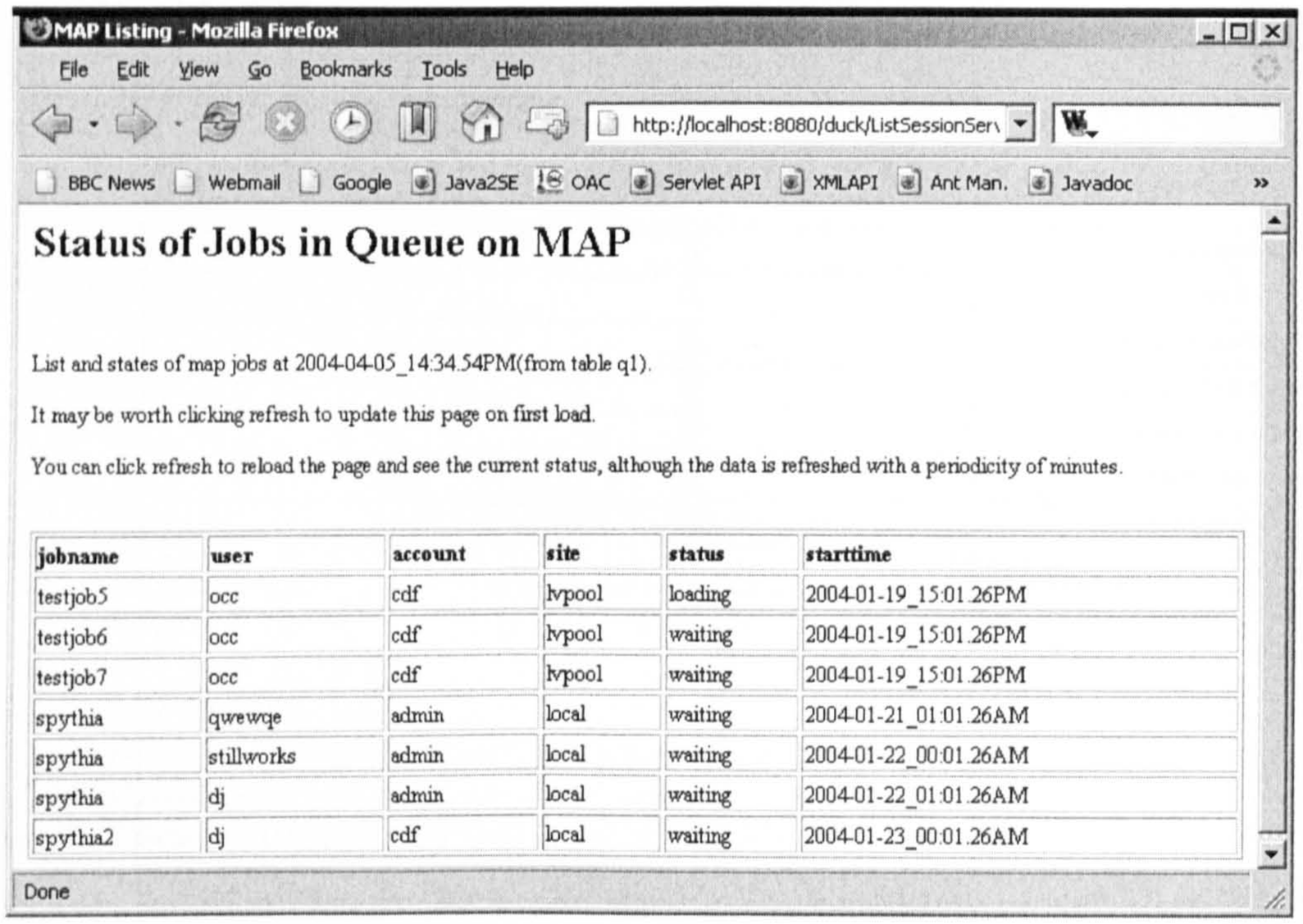

Figure 8.12: The web page that allows users to check the job queue.

8.5 Deployment

The DUCK web application was deployed on the Apache Tomcat Server. The Apache software foundation is a non-for-profit corporation which produces open source software. The Tomcat server is part of the Jakarta project and is a *servlet container*. It is the official reference implementation for Java servlets and JSP. There are three generations of Tomcat server, these are;-

- 5 The latest version of Tomcat, supporting Servlet Specification 2.4
- 4 An older version of Tomcat, supporting Servlet Specification 2.3. This version implemented a new servlet container called Catalina.
- 3 The oldest currently supported version of Tomcat, supporting Servlet Specification 2.2. This version of Tomcat has roots back to the original release of Apache Tomcat.

The Apache Tomcat version (4.1.29) was used; this version was the latest version of the Tomcat 4.1.xx series available at the time. The web application was deployed on Windows and Linux systems.

8.5.1 Servlet Container Configuration

The web application is configured as such using the standard `web.xml` file. This file is an XML document that is described by the *document type definition* with the namespace `http://java.sun.com/dtd/web-app_2_3.dtd`. The document contains the following information;-

- The *fully qualified classname* of every servlet in the web application
- A name for each servlet
- A *URL pattern* for each servlet, that defines the location of the servlet in terms of a relative URL
- A *display name* for the web application
- A *description* of the web application, a human-readable description of the web-application.

The `web.xml` file has the root element `web-app`. `Servlet` and `servlet-mapping` are two possible tags that can be contained in the root element.

The name and qualified classname for a servlet are contained in `Servlet` elements, whilst the URL pattern is contained a `servlet-mapping` element. The order of the elements tends to be important; all the `Servlet` elements should be grouped together and all the `servlet-mapping` should be grouped together. The two types of elements should not be intermixed.

8.5.2 Windows

Deployment on Windows involved copying the completed web application archive (WAR) or the files in the correct paths to the correct folder in the

Tomcat Installation and restarting Tomcat. If not previously installed, the MySQL database system would need to be installed and configured.

8.5.3 Linux

Installing the Web-application under Linux followed a similar procedure to Windows, although due to differences in where the Java property user.home points to, the properties.txt file needs to be placed in a different location.

8.6 Starting a Job using DUCK

The initial step for a user would be to point his/her browser to the URL, `http://localhost:8080/duck/` (in the case of the test version).

The browser is supplied with a static page which is not generated by a servlet. This contains a form to allow the user to provide the login details (see 8.8). The user fills in his/her login details, which consist of the account name and the user name. MAP has the concept of accounts, which are groupings of users corresponding to the experimental groups, for example LHCb, CDF and ATLAS. The username is individual to the user. When the user clicks 'Login', the browser dispatches the details, via the POST method, to the `DuckID` servlet (see 8.4.1).

The functionality of `DuckID` is also covered in 8.4.1. In the response `DuckID` sends a cookie for the browser to store. This cookie contains a unique ID (session ID) that links the user with the data held in the DUCK database. This ID is simply a random number of type long. The HTML response contains a web page with links that allow the user to do a number of tasks (see 8.9).

To submit a MAP job the user should click Job Control Block Entry. This returns a page illustrated in 8.10. The user will then fill out the Job Control Block parameters, these are parameters that relate to the computing side of the job (see 8.3.6). When the user clicks 'SubmitParameters', the form data is POSTed to the `JCBServlet` (see 8.4.1).

`JCBServlet` retrieves the session ID from the cookie in the browser request. This allows DUCK to store the Job Control Block parameters the user has entered referenced against to the session ID in the database.

`JCBServlet` also checks the values entered into the submitted page, mainly to check the values of the right type. Once the values are stored the servlet returns a web page with a link to the next servlet.

This webpage automatically redirects to the `JobParaServlet` (see 8.4.1). This servlet reads a set of parameters describing the physics of the computing job (for a monte-carlo simulation). More specifically the parameters allow differences to be introduced to the jobs running on the nodes. Otherwise there would be, for example, 300 identical jobs running. The parameters and their default values are stored in a database table. Each setup job has its own database table holding the parameters for it.

The parameters and their default values are returned as a form in a webpage (see 8.11). This gives the user a chance to edit the values. The 'Submit Parameters' button sends the values to the `SubmitJobServlet` (see 8.4.1).

The `SubmitJobServlet` retrieves the Job Control Block and submits it along with the job parameters to the MAP core system using a network socket.

The MAP system returns, as a response, a possible series of *errors*, *warnings* and *information*. See section 8.3.8 for a description of the meanings of these categories.

If there are no errors the `SubmitJobServlet` includes a link allowing the user to submit the job. The users response is handled by `ConfirmServlet` (see 8.4.1) which opens a socket to the core MAP system and signals the job should be submitted to the job queue.

8.7 DUCK as Grid Middleware

A number of solutions to 'gridifying' MAP were considered. One solution involved using a system that was developed to maintain the GridPP website (see

[70]). Using *GridSite* allows grid security to a web-based system is described and offer in [71].

Another more mainstream solution is to use the Globus toolkit (see [20]). This was the solution decided on.

Some work to implement a grid service was done using Globus OGSA (Open Grid Services Architecture) 3.2. The Globus Toolkit 3 has bindings to allow grid services to be implemented in Java. This was a major factor in the choice of Globus; existing Java code from DUCK could be used to implement a grid service.

The code implemented amounted to the implementation of the session ID system (see 8.4.1) as a grid service. The toolkit's Java API's were used, so that the existing DUCK was useable in a grid solution. Client-server communication is provided by a web service like system in Globus Toolkit 3. By using the tools provided XML schemas and Java bindings for the services were easily generated. In this test software the client and server were both command line software. This Work was not continued due to time constraints.

One problem, which would be solved by using a grid solution, was the uploading a large files and even data sets. Replica managers and grid-enabled FTP services offer ways to provide this functionality.

8.8 Summary

This chapter covered, in detail, the software that made up the web-interface to MAP, this software consists of a web application written in Java. The many design decisions, interfacing to the core MAP system and deployment were all discussed. A relational database system provided persistency and data storage for the web application. In addition an XML based system provided a settings mechanism.

DUCK was deployed on the original MAP 1 system and used to submit computing jobs. The system was found to function correctly and greatly eased the

submission of jobs to the MAP system.

Chapter 9

Conclusions

Spitfire Web was a proof-of-concept *web application*, its implementation taught me a great deal about technologies such as the Java platform, servlets, JDBC, relational databases and web services.

A number of design decisions were made for me, due to having to interface pre-existing code. A major design decision, already made, was the use of Java which provided platform independence. The use of SOAP as the language of the messaging system had already been decided on. JDBC was a natural choice for the proof-of-concept messaging system. A final major choice was the use of servlets, and hence a web-interface.

I had more choice deciding the following; the use of a single servlet (I followed a different direction when writing DUCK) and multiple support classes. URL re-writing was chosen as the persistency solution. The use of Javascript allowed certain pages in the interface to be interactive. Finally, ANT was chosen as a build tool to compile and deploy the web application The completed solution worked effectively using the JDBC interface and showed how a future version could work.

DUCK was developed to allow users that were not experts in the MAP system to submit jobs. To do this a web-interface was selected as the user interface. This simplified the matter of providing a client; a standard web browser is sufficient.

Java was chosen as the language in which to implement the project in. This was a departure from the core MAP system, which is written in C. Again, servlets provided the web interface but this time the web application consisted of multiple servlets.

The interface between the DUCK system and MAP was jointly designed around network sockets. Sockets were chosen because they represented a simple way to interface between the C code of the MAP system and the Java of the DUCK system.

Again the build tool ANT was used to compile, document and deploy the web application.

Persistency for the web application was provided by a database system which used cookies to identify users. XML was used to provide a generic query system to allow various views of MAP data to be produced. The use of such widely used technologies allows the system to be easy expanded at a later date. This expansion could be interfacing with different types of computer cluster or using the DUCK web interface to MAP to provide a solid base for turning MAP into a *grid resource.*

The completed system was used to start Monte-carlo generation runs for the CDF detector group.

Appendix A

Software used list

Here is a list of the packages used to develop the software described in this thesis. Usually the latest stable version of any given package was used.

- Netbeans Integrated Development Environment (see [33]). This package was used to edit Java source code. Version 3.6 was used.
- JEdit (see [72]) and Crimson Editor (see [73]) were two text editors I used extensively.
- Standard Edition Java Development Kit (see [74]), compiler, JVM and tools. I used a number of 1.4.xx releases.
- ANT was used for builds (see [32]).
- The Mozilla Firefox, Netscape and Microsoft Internet Explorer browsers were all used for testing.
- MySQL 3.x Database servers and clients (see [58]) were used to provide Relational Database services. The MM MySQL driver was used as a JDBC driver (see [75]).
- Apache Tomcat Server, [66], (version 4.1.xx, typically 4.1.29) was used to deploy the software for testing and use. Tomcat is an example of a *Servlet Container.*

Appendix B

Dublin Core Metadata Initiative

The *Dublin Core Metadata Initiative* (DCMI) [35] promotes the adoption of common and interoperable metadata standards. The DCMI has defined a set of metadata standards, these were devised by consulting with representatives from a large number of fields. One such principle is to promote interoperability between knowledge domains (metadata sets), via frameworks, specially designed for this purpose. The DCMI encourages communities to develop metadata themselves to describe their own knowledge domains.

The DCMI has defined a minimal metadata set or schema to be used in the context of the world wide web environment. It consists of 15 fields and resides in an XSD schema (see C.2), these fields/elements, divided into 3 categories:-
Content

- Title
- Subject
- Description
- Type
- Source
- Relation
- Coverage

Intellectual Property

- Publisher
- Contributor
- Rights
- Creator

Instantiational

- Date
- Format
- Identifier
- Language

These fields are provided so that document creators, or those who catalogue documents, can provide a set of information allowing items to be 'discovered' over a network in an automated fashion.

Appendix C

XML Related Specifications

C.1 Namespaces

Namespaces are a mechanism to solve the problem of ambiguity between symbols (elements) with the same names. These problems also prominent in programming languages; C++ makes use of *namespaces*, whilst Java has an equivalent feature, *package names*, which provides a solution to the problems arising from ambiguity.

For example; one person could define an element called

$$\langle address \rangle$$

with the intention of it storing the postal address of an individual or company. Whereas someone else could mean it to define a web address, a URL.

Namespaces solve the ambiguity by using identifier elements that define different namespaces. Hence, the two address elements could become;-

$$\langle web : address \rangle$$

$$\langle postal : address \rangle$$

Which would imply one element belonged to a namespace being identified as web and the other one in a namespace identified as postal. The above

syntax is known as qualified name syntax. The postal namespace could have other elements and attributes and the existence of the namespace would help distinguish these from elements in the namespace postal, even if there were no ambiguities.

To use a namespace in a document it must be declared first. The declaration takes the form of an element with attributes, any child attributes of this element are in the namespace declared. Such an element could be the root element, for example. The attribute takes the form of:-

- xmlns:namespaceprefix=url.

Where:-

- xmlns is a keyword to show the attribute is a namespace declaration.

- namespaceprefix is the name space prefix, used to prefix the elements in the namespace.

- url is a URL belonging to the maintainer of the namespace. XML processors are not required to do anything with this URL, although the maintainer could provide information about the namespace at this URL.

XML schemas depend heavily on namespaces.

C.2 Schemas

Document Type Definitions (DTD) are used to define the document structure in HTML and are related to SGML. HTML is another user of DTD, which act like a blue print for a document; it defines a list of legal elements for the document. They can also be used to verify existing documents, checking they conform to what the DTD expects.

Despite the wide spread use of DTDs they have a number of limitations, especially for use with XML. Firstly, DTDs are not XML, therefore the tools used for parsing XML cannot be used to parse DTDs. Secondly, whilst DTDs define the structure of documents, they don't have any real means of defining types. Typically in XML applications, you have quantities such as integers and types such a strings and dates. Also, there is a requirement to define your own data types; this is done using XML schemas; these act as a blueprint for a document. The schema can specify whether a types expected or required and whereabouts in the document it is needed.

To overcome the limitations of DTDs the W3C proposed XML schemas. The standard is composed of two specifications; the first *XML Schema Part 1*, deals with structure and the second, *XML Schema Part 2*, deals with data types. Both parts are now W3C recommendations.

As XML schemas are valid XML documents the same tools that are used to process other XML documents can be used to process them. XML schemas also provide a rich set of datatypes, to which custom complex types can be added.

Using XML schemas we can define a documents form, in terms of structure and the data it should contain. XML parsers can check instances of documents against their schemas, validating them in the process. Validated documents are those that conform to their schemas. XML schemas act as specification for documents; using a schema one can work out how to read or write an instance of a document or even how to convert it into a different form. Hence, different forms can be used to represent data. These forms can be data that of a more distilled form (for example more relevance, filtering or processing) or simply different presentational forms (for example portable document format, Microsoft Word, HTML). This allows the interchange of information. Information interchange is an issue especially in large corporations.

There are other schema languages aside from XML schemas, an example of which is Relax NG Schema[76].

C.3 Xpath

XPath is a standard that allows parts of XML documents to be referred to. Paths are defined to locate elements in the hierarchical structure of XML.

C.4 Xlinks

This specification allows XML documents to refer to other XML documents and external resources. Xlinks are somewhat analogous to hyper links in HTML, although complex relationships can be defined.

C.5 Xpointers

XPointers allow specific portions of XML documents to be referred to from external documents; a combination of Xpaths and Xlinks.

C.6 XLST and other Technologies

eXtensible Style Language Transformations (XSLT) are instances of the eXtensible Stylesheet Language (XSL). Both languages are recommendations of the world wide web consortium. XSL allows stylesheets to be defined in XML.

In HTML stylesheets are used to format the appearance of documents; the aim of this is to separate the document content from the presentation information. The motivation behind this is to allow a single document to be presented in a number of different forms using different stylesheets. For example, a document could be processed to form a web page, a printer friendly web page, a WML document, a Portable Document Format file and many more formats, all using stylesheets.

In HTML stylesheets are defined by the Cascading Style Sheet (CSS) language. CSS has been relatively successful as a means of defining document appearance

in HTML (despite patchy support from Browser Vendors).

Whilst CSS was extended for XML a more extensive solution was required in the form of XSL. XSL is an XML language so some of the arguments in the DTD versus XML schemas debate are applicable to the CSS versus XSL argument. XSL goes further than CSS to provide scripting allowing XML documents to be manipulated as well as defining styles. Being a relatively large language, XSL is split into 3 parts;-

- XSL-FO (XSL Formatting Objects), a markup language, purely dealing with presentational details.
- XSLT (XSL Transformations), a markup language. This is for transforming an ML-based markup language into other markup or text.
- XPath (XML Path Language). This is used in conjunction with XSLT to point to information with XML documents.

Using these three languages we can take an XML document and related stylesheet and create a new document using an XSLT processor.

Typical applications of this process are listed below:-

- Convert an XML document into HTML. This allows information to be viewed easily in a browser The process can be done server side or client side, although support for this is limited
- Convert XML documents into other Markup languages such as WML (Wireless Markup Language)
- Extract information from existing XML documents
- Convert XML documents to different vocabularies.

An XSLT processor is a software component that can perform the requirements of the XSLT standard. Examples of XSLT processors are Xalan (Apache Software Foundation), Microsofts XLST Processor, Oracles XLST Processor and James Clarks XSLT Processor.

Appendix D

SQL

Structured Query Language (SQL) is not just a query language; it is also a *data definition language* (DDL) and a *data manipulation language* (DML). SQL is cited as one of the major reasons for the success of relational database systems because it provides a common standard for relational database systems. For example, if a user were to become dissatisfied with the DBMS that they are using they could migrate to another DBMS with little difficulty.

The SQL language is designed so that users without programming knowledge can use it to formulate queries.

SQL was originally known as SEQUEL when it was developed at IBM Research as an interface to their System R relational database system. ANSI (American National Standards Insitute) standardised SQL in 1986, producing something known as ANSI 1986 or SQL-86 or SQL1. A revised standard was published in 1992 (SQL-92 or SQL2). The latest standard, SQL3, has iterated through 2 versions (at time of writing). The various ISO standards can be found on the ISO website[41].

Like most successful IT standards, the functionality of the SQL implementations can be divided into two areas, one being the SQL functionality implemented by all vendors, and the other being the functionality that vendors tack onto the standard, potentially making their implementation not-interoperable with other vendors.

D.1 Data Definition Commands

SQL provides commands for data definition; `Create`, `Alter` and `Drop`. These three commands tend to be used with the keywords *table* or *schema*, allowing the operations to be carried out on tables or schemas. To create a table, information that defines the names and data types of the fields is needed for each field (or column) of the database table. To use the *drop* command the table name needs supplying, this then results in the table being dropped. The *alter* command is used to alter database tables. This is usually to add or drop columns, change the data type of a column or to change a constraint.

D.2 Query Commands

SQL has one query command; the `select` statement. The basic form of this statement is as follows;-

```
SELECT [attribute list ]FROM [table-list] WHERE [condition ];
```

where:-

`[attribute-list]` - is the attributes or fields that are to be retrieved, sometimes the field names need to be qualified with a table name as the same field name can exist in different tables. The use of * instead of the attribute list means all attributes are retrieved. Using the keyword `DISTINCT` means duplicate results in the result set are deleted. The use of the keyword `ALL` means these duplicates are not deleted.

`[table-list]` - is a list of relational tables that are needed in the processing of the query

`[condition]` - is an expression that evaluates to a boolean value. Two conditions can be specified by using the AND keyword. Appending `ORDER BY [attribute list]` - allows the results of the query to be sorted into an order using the specified attributes.

D.3 Data Modification Commands

Databases in SQL can be modified using 3 commands; `Insert`, `Delete` and `Update`. The `insert` command takes the following form:-

- `INSERT INTO [tablename]VALUES [values]`

where: -

- `[tablename]` - is the table to insert records into
- `[values]` - are sets of values to insert as records.

Multiple records can be inserted in one statement.

The `delete` command has the form:-

- `DELETE FROM [tablename]WHERE [condition]`

where: -

- `[tablename]` - is the table to delete the records from
- `[condition]` - selects which records to delete.

The update command has the form:-

- `UPDATE [tablename]SET [field, value pairs]WHERE [condition]`

where:-

- `[tablename]` - is the table in which records are to be updated
- `[field,value pairs]` - are the fields to update with new values
- `[condition]` - condition selects which records to update.

D.4 Privilege Commands

The SQL privileges system controls access to databases; the commands `Grant` and `Revoke` are central to this privileges system. To use the database system a person must be first given a user account on the system. One example of an account would be the web-user account that a database system running behind a web server would have. In this case all users accessing the database over the web might use one user account.

Different user accounts will have different amounts of privilege. The web-user account described above would probably have limited privileges, with for example, no ability to modify or add data and the ability to only view a subset of the data.

The ability to define the access permissions for databases is especially important in situations where sensitive information is stored, for example banking and medical records.

Whilst the user is using the database, the system records what the user does in a log. This log can then be used to provide an audit trail; allowing their actions to be reviewed.

Privileges can be assigned to users (at the account level) or to tables (at the relation level).

Appendix E

JDBC Driver Types

E.1 Type I JDBC Drivers

Type I category drivers use ODBC to access databases. In some cases this may be the only available solution if there is no JDBC driver for the database. A ODBC-JDBC bridge driver comes with Java. The servlet (or program) will use this to communicate with the ODBC API, which in turn uses an ODBC driver to make calls to the database. With all these layers, this is not an efficient solution. Functionality is also limited to what the ODBC driver provides.

E.2 Type II JDBC Drivers

A type II driver will use part Java, part native code driver to communicate, via a vendor specific protocol (using a library), to the database. The driver and library are on the client. The response (the results) will be formatted to JDBC standards by the driver and returned to the program.

E.3 Type III JDBC Drivers

This category of driver is also known as a net-protocol driver and has three tiers. The client sends JDBC calls to a piece of middleware on a server. This is done using pure Java. The server then converts the calls to the vendor dependent access calls and communicates with the database server. These types of drivers have better performance than type I or II and allow for extra functionality, such as encryption, to be used. The downside is the middle server, which creates extra installation work, can degrade performance and must be running for access to the database to be possible.

E.4 Type IV JDBC Drivers

This is the preferred solution to connect to a database. There is a pure Java JDBC driver, which uses a vendor protocol to connect directly to the database. There are no native libraries to worry about, hence, the drivers are usually jar files

[1], which are easily installed. The client communicates to the database server using a native protocol from the pure Java driver.

[1] *Jar files* are compressed archives containing Java class files (programs), their name is derived from *Java ARchives*. The format is similar to a zip file and indeed a Jar file can be created in the same way, although this compatibility is not guaranteed in the future. In addition a Java Archive has a *manifest* describing what it contains.

Appendix F

Hardware Elements

Some elements of hardware are briefly described below. The site [77] provides a good starting point for detailed descriptions of how hardware works.

F.1 Circuit boards

The motherboard or mainboard is a central component of a PC. It consists of a multi-layer circuit board that has the following items on it; -

- A system Basic Input Output System (BIOS), the basic control program for the system
- A socket or slot to hold the CPU
- Slots for memory modules
- Expansion slots for connecting expansion cards, such as network cards and video cards.
- Connectors for the low voltage power supply to the motherboard
- An I/O panel for keyboard, mouse and other connectors.
- Some modern boards have drive controllers, video, sound adapters and various other functions integrated on to the board.

Data buses on the motherboard are responsible for transferring data between the different devices connected to it. Timing and other problems associated with fast buses over long distances, there are two different speed buses on motherboards. One is a high speed bus that connects the CPU, cache memory and main memory and the other is a slower speed bus is used to connect other devices such as expansion cards.

F.2 Data Storage

F.2.1 Hard Disks

Hard disks store data using magnetic flux on a spinning media. Data is written to a magnetic substrate on a spinning platter by heads. Virtually all hard disks have multiple platter and head sets contained in them. These platters spin at rates of around 3000 to 10000rpm on the fastest drives. The gap between the heads and the platters is extremely small; dust particles are massive compared to the tolerance. This means the drives have to have extremely clean interiors. The precision of the drives allow them to store orders of magnitudes more data than floppy disks and access the data at much faster rates.

There are two common interfaces to allow computers to access hard disk; -

- *Integrated Drive Electronics* (IDE). Most of the electronics are integrated onto the drive assembly in the form of a circuit board. IDE drives have the larger market share and can be considered commodity hardware.

- *Small Computer System Interface* (SCSI). This interface standard was designed to allow many different devices to be connected using it. SCSI devices tend to run at higher data transfer rates and are more expensive. The interface tends to be used on high end machines and servers.

A technology known as *Redundant Array of Inexpensive Disks* (RAID) is used to create arrays of multiple hard disks. RAID can be used to either increase

redundancy or increase performance. By duplicating amounts of data to multiple disks in the area, data can be recovered if a disk fails. Other techniques can be used to increase performance, in addition RAID offers at least 9 levels, all used for different purposes.

F.2.2 Tapes

Tapes provide high capacity data storage; information is written onto a tape coated with a magnetic substrate. Tapes can be written many times and read many times. The access time tends to be a lot slower than hard disk and quite often a large proportion of the tape needs to be read to retrieve small amounts of data.

Tape robots can be used to allow a large number of tapes to be accessed automatically.

F.2.3 Optical

Data can be stored on optical media. Optical disks have the data stored on a continuous spiral track. The information is written and read using a laser. For high capacities it is possible to build CD-robot that have large capacities of data online, in the same way in which tape systems work. Of course this type of system is slower than having arrays of hard disk constantly online. There are also question marks over the lifetime of the media, this has repercussions in terms of data loss.

F.3 Commodity

The term *commodity* is synonymous with items that are readily available, cheap, compatible with other systems and not proprietary.

Long before IBM-compatible PCs emerged, nearly all systems were proprietary. The hardware, the software and operating system were produced by the same

company; incompatible with other manufacturers systems.

Control Program for Microcomputers (CP/M) was a turn in this trend. CP/M was an operating system used in the 1980's, which ran on most micro computers that had disk drives. At the higher end of the market (servers and work stations) UNIX appeared.

When IBM created the PC they decided to publish the hardware specifications and use an operating system provided by another company (Microsoft). These two decisions started the formation of a commodity market for PC hardware and, to some degree, a commodity market for software.

By publishing the hardware specifications for their PC, IBM intended to make to easy for vendors to create expansion cards for the system. They did not foresee that the entire PC system, from main board to basic system would be cloned. Intel were the producers of the CPU, so soon it was possible to buy an IBM-compatible PC with no IBM parts in it. This situation has led to the commoditised PC market; with a choice of vendors for almost all the components that build up a PC.

Appendix G

Software

G.1 Operating Systems

The operating system is software on a computer that manages the operation of the computer system. It does this by interacting with hardware, managing interaction with such systems as memory, disks and application software. Additionally, operating systems are responsible for task scheduling and interacting with the user.

By providing this functionality in the operating system application programmers do not need to worry about providing it. Instead they use libraries of functions to access the services provided by the operating system.

The major operating systems currently are the UNIX/Linux based family (which includes Apples operating system) and the Microsoft Windows family.

Unix

UNIX was originally developed by employees from ATand T and Bell Laboratories and is a POSIX compliant operating system. The UNIX operating system supports multi-tasking, multiple users and is implemented in the C programming language.

UNIX and its fore-runners have a long and complex history dating from the late 1960's. It has traditional been a popular operating system in academic and industry environments.

By introducing many innovations UNIX has had a large impact on the computing community. UNIX has provided a hierarchical file system into which devices and services fit neatly. The operating system also gave developers early access to the TCP/IP networking standard (see 5.2.2); cementing its use for network applications.

Linux

Linux is a UNIX clone written by Linus Torvalds in the early 1990's. He then invited the global community of developers to contribute to it. It is subject to the licensing of the GNU (GNU is Not UNIX) project, which ensures users have certain rights with respect to the software it licenses. Originally developed for the Intel 80386 CPU, Linux has been ported to many other operating systems.

Like UNIX, Linux is written entirely in the C programming language. Linux tends to be distributed as a distribution, which consists of the core linux software (the kernel etc), libraries and end user-applications.

The title [78] covers some of the technical issues involved with programming Linux.

Windows

Microsoft Windows (see [79]) is a proprietary operating system with nearly all the market share for operating system software. The system is based around a Graphical User Interface (GUI) and was developed for IBM-compatible PCs. Windows has evolved from a GUI layer, which ran on top of Microsoft Disk Operating System (MS-DOS). MS-DOS in turn was bought from a vendor who wrote it borrowing many ideas from CP/M.

The MS-DOS/Windows line of operating system has been developed to create

Windows 1,2 and 3, all of which were layers on top of MS-DOS. Starting with Windows 95 the presence of MS-DOS became less obvious in the line of products. Windows 95 moved the product line from 16 bit to 32 bit and offered a more complete user interface. Windows 95 was replaced by Windows 98 and finally Windows ME, neither product offering any major innovations over Windows 95.

Based on its work with IBM, Microsoft developed the Windows NT line of operating system. Until the advent of Windows XP, this line of operating system was sold along side the MS-DOS based version of Windows. The Windows NT line, also marketed as Windows 2000, offered much greater reliability and more UNIX like features. With the release of Windows XP, based on the NT code base, the NT based line of products replaced the MS-DOS based line of products.

Windows NT has been ported to other some architectures than the Intel/IBM compatible PC platform. The Windows platform has been the target for many security exploits, partly due to flaws in the system and partly due to its popularity.

Mac OS

The current Apple operating system, OS X, is based on UNIX. This makes OS X one of the most widely deployed UNIX-variants around. Before the arrival of OS X , Apples operating system was a proprietary system.

Appendix H

Cluster Types

The following paragraphs provide a brief look at a few different cluster-based systems, it is not exhaustive. *Beowulf* tends to refer to the hardware to build a cluster, Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) refer to software and software standards.

H.1 Beowulf

Beowulf Cluster is a term used to describe computer clusters built from commodity hardware and which use Linux based software to control the clusters and run applications. The clusters run on private networks and are used to solve high performance/parallel computing tasks. The concept of Beowulf clusters dates from the early 1990's at NASA, spreading through academic circles rapidly.

It is argued by some that to give the designation Beowulf to a cluster it has to use entirely commodity hardware. This would apply to class I clusters. Class II clusters use specialised hardware to gain greater performance.

Typically the software for Beowulfs is written in C or Fortran. There is no one piece of software that is known as 'Beowulf'; there are many Linux components that can be used to provided the required software for running a cluster. The

Scyld Beowulf distribution is designed for the purpose of running Beowulf clusters. See [80] for more information on this type of cluster.

H.2 MPI

MPI is a user community developed standard. It is a protocol designed for parallel machines and work station clusters. MPI defines a format for the passing of messages between machines in a cluster. Typically applications using MPI run in the following way;-

- The machines are set up with the executable applications; usually identical on each machine

- Each machine uses the same program, typically, to solve a small part of the problem. A good example is an aerodynamic simulation of a surface moving through a volume. The volume is split into cubes and each machine does calculations on a cube.

- Initial conditions are set to each machine

- Calculations are done on each machine; this is the processing phase

- All the machines stop calculating and exchange boundary conditions on their cube, this is the messaging phase

- Calculation resumes

- The process continues until the simulation is complete.

The MPI standard consists of the older MPI 1,1.1 and the newer MPI 2. MPI specifies processes for message passing, which includes, point to point communication, language bindings, methods for process creation and management and management and discovery of the operating environment. See [81], [82] for more details.

H.3 PVM

PVM is a package to create a large parallel computer from heterogeneous nodes. These nodes can be running either Unix or Windows based operating systems and are connected over a network. PVM supports different types of networks. The package supports heterogeneous node hardware in two ways for applications;-

- Differences can be masked from applications
- Differences (such as some machines having dual processors) can be provided so that applications can exploit them.

The pool of nodes can have machines added and deleted to it during the running of a job. This provides fault tolerance.

The PVM package is divided into two parts; -

- pvmd, the PVM daemon which resides on all nodes
- PVM library, a library of interface code for PVM.

See [83] for more details.

Appendix I

RPC

Remote Procedure Call (RPC) is a protocol allowing remote procedure calls, meaning a program running on one host can invoke code on another host (using the network). RPC allows the programmer to code remote calls in a fashion that is similar to local calls. RPC uses the client-server model is used by RPC with the client initiating the RPC by sending a request the remote machine.

Transport independence is achieved by using RPC and it is a popular method for constructing distributed application software.

A number of RPC implementations have been produced. A successful RPC was written by Sun Microsystems. It provides the basis of many network services including *Network File System* (NFS). Portmapper is another RPC based tool.

Other technologies provide alternative protocols to RPC for distributed computing. A current trend is the use XML-RPC as a transport media for RPC.

I.1 DCE

Distributed Computing Environment (DCE) is the packaging of various components, including RPC, to provide an environment for distributed computing. The components are:-

- Remote Procedure Call (RPC)
- Cell Directory Services (CDS)
- Global Directory Services (GDS)
- Security Service
- DCE Threads
- Distributed Time Service (DTS)
- Distributed File Service (DFS)

The following are known as secure core components and are required on any installation; DCE Threads, RPC, CDS, Security Service and DTS.

There are varying amounts of supports for DCE on differing platforms. Some platforms only support the secure core (the minimum required for DCE), some support only clients and other platforms support the entire environment.

DCE is aimed at procedural programmers and is not fully Object Orientated. See [84] for more information about DCE.

I.2 CORBA

Common Object Request Broker Architecture (CORBA) [85] is a framework produced and maintained by the *Object Management Group* OMG). CORBA allows distributed applications to be written that are platform independent.

CORBA is useful for the following reasons; -

- It can broker between many different machines running different platforms. Examples of the diversity of these platforms range from mainframes, to desktops to hand held PCs.
- It can be used to construct servers that are about to handle a large number of clients, with high rates of request.

- It possesses reliability and scalability.

CORBA allows programming languages to access objects written in different languages. For example a Java program could access a C++ object in another program, even on another machine.

Interface Definition Language (OMG IDL) is completely language independent and has to be mapped to a language. It has been mapped to many programming languages including; C, C++, Java, COBOL, SmallTalk, Ada, Lisp, Python and IDLScript.

Like Java RMI (see section I.5) CORBA uses a client-server pattern with stubs and skeletons. The stub is found on the client and is the point from which methods are invoked. These method calls are wired across the network to the server, where they are matched up with the methods in the skeleton and the correct method calls performed on the server. CORBA uses Internet Inter-Orb Protocol (IIOP) to transmit its requests across the network. IIOP can be used to transmit RMI (see section I.5) calls as well.

To allow the system to work with different programming languages CORBA uses IDL to define the stubs and skeletons; the programmer must write the bindings in terms of IDL. Applications in CORBA consist of objects; usually many instances of these objects exist to represent entities, for example a shopping cart on an e-commerce website.

Legacy applications, for example, an accounts system, can be wrapped with a single instance. This is generally the most typical approach.

I.3 COM

Component Object Model (COM) technology is a Microsoft architecture designed to allow applications to be constructed from binary components. This means basic components can be written in a COM-compliant fashion and used to construct higher level software services. ActiveX and OLE (Object Linking and Embedding) are technologies that are implemented using COM.

I.4 DCOM

DCOM (Distributed Component Object Model) is an extension of COM, allowing technology to work over a network. It exists on the Windows platform but has thus far failed to move to other platforms. For an architectural view of DCOM see [86].

I.5 RMI

Remote Method Invocation (RMI) is a Java-specific way to do distributed computing; meaning whilst it is not language independent it is platform independent.. RMI allows class methods to be called on a remote Java Virtual Machine. The system works in such a way that once the reference to the remote object is obtained, working with the remote class is the same as working with an 'normal' local class.

To make a class a remote class it has to implement the interface javax.rmi.Remote. Implementing the class implies that the object can be invoked remotely. Once this remote object has been written and compiled a tool is used to generate a skeleton and stub from it.

The client uses the stub to provide a local interface locally. The client calls the stubs local methods; these method calls are passed over the network to the server. On the server the methods calls are reconstructed by the skeleton as local method calls on the server. The client sees the method calls as local method calls. Any exceptions occurring on the server are passed back to the client via the stub.

In order for the whole system to work an RMI registry must be running on the server. This allows clients to obtain references to the remote objects running on the server. Typically port 1099 is used for the network port. Over the wire RMI uses the protocol Java Remote Method Protocol (JRMP), which is also known as an RMI wire protocol. Although this is the main protocol used it is possible to use other wire protocols with RMI.

Often restrictions are encountered with firewalls that stop RMI working, making its usage in an Internet environment difficult. For a more detailed discussion of this topic see [87]. For a comparison between CORBA, DCOM and RMI see [88].

I.6 SOAP

Simple Object Access Protocol (see section 5.5.4) is an example of XML-RPC, which allows remote procedure calls to be exchanged using XML as the messaging media. Using SOAP allows loose coupling; meaning it is not tightly bound to any one language or platform.

Bibliography

[1] Various, "The CERN Website", *CERN, http://www.cern.ch/, 2005* .

[2] Various, "LCG - LHC Computing Grid Project", *CERN, http://lcg.web.cern.ch/LCG/, 2005* .

[3] Ian Foster, Carl Kesselman, "The Grid: Blueprint for a New Computing Infrastructure", *Morgan Kaufmann, San Francisco, USA, 1999* .

[4] Rosy Mondardini, "Grid Cafe", *CERN, http://gridcafe.web.cern.ch/, 2005* .

[5] Christos J.P. Moschovitis, Hilary Poole, Tami Schuyler and Teresa M. Senft., "History of the Internet, 1843 to the present", *Moschovitis Group, http://www.historyoftheinternet.com/, 1999* .

[6] W3C, "About the World Wide World", *W3C, http://www.w3.org.org/WWW/, 1992* .

[7] Various, "Enabling Grids for E-Science", *http://public.eu-egee.eu/, 2005* .

[8] Various, "CERN Batch Services", *CERN, http://batch.web.cern.ch/batch/, 2005* .

[9] I. Foster, C. Kesselman, S. Tuecke., "The Anatomy of the Grid: enabling Scalable Virtual Organisations", *Globus, http://www.globus.org/alliance/publications/papers.php, 2001* .

[10] Various, "ALICE A Large Ion Collider Experiment at CERN LHC", *CERN, http://aliceinfo.cern.ch/, 2005* .

[11] Various, "Atlas Collaboration", *CERN, http://atlas.web.cern.ch/Atlas/index.html, 2005* .

[12] Various, "CMS Compact Muon Solenoid", *CERN, http://cmsinfo.cern.ch/Welcome.html, 2005* .

[13] Various, "LHCb Home Page", *CERN, http://lhcb.web.cern.ch/lhcb/, 2005* .

[14] Various, "The Globus Alliance Website", *The Globus Alliance , http://www.globus.org/, 2005* .

[15] Various, "Condor Website", *University of Wisconsin Madison, http://www.cs.wisc.edu/condor/, 2005* .

[16] Various, "Legion Website", *University of Virginia, http://www.cs.virginia.edu/ legion/, 2005* .

[17] Ian Foster, Carl Kesselman, S. Tuecke, "The Anatomy of the Grid", *Globus Alliance, http://www.globus.org/alliance/publications, 2001* .

[18] Various, "Platform Computing LSF ", *Platform Computing, http://www.platform.com/, 2005* .

[19] Various, "Open PBS", *Altair Grid Technologies, http://www.openpbs.org/, 2003* .

[20] Various, "Globus Website", *Globus, http://www.globus.org/, 2005* .

[21] Mike Neuffer, "Linux High Performance SCSI and RAID", *http://www.staff.uni-mainz.de/neuffer/scsi/, 2005* .

[22] Ian Foster, Carl Kesselman, Gene Tsudik, Steven Tuecke, "A Security Architecture for Computational Grids", *Globus, http://www.princeton.edu/ rblee/ELE572Papers/p83-foster.pdf, 1998* .

[23] Nataraj Nagaratnam, Philippe Janson, John Dayka, Anthony Nadalin, Frank Siebenlist, Von Welch, Ian Foster, Steve Tuecke,

"The Security Architecture for Open Grid Services", *Globus, http://www.cs.virginia.edu/ humphrey/ogsa-sec-wg/, 2002* .

[24] Joel Weise, "Public Key Infrastructure Overview", *Sun Microsystems, http://www.sun.com/blueprints/0801/publickey.pdf, August 2001* .

[25] Various, "x509", *Open SSL Project , http://www.openssl.org/docs/apps/x509.html, 2005* .

[26] Various, "Security and Authentication", *National Centre for e-Social Science , http://www.ncess.ac.uk/grid/security/, 2004* .

[27] Chuck Cavaness, Geoff Friesen and Brian Keeton, "Using Java 2 Standard Edition", *Que, Indiana, USA, 2001* .

[28] Brian Bramer and Susan Bramer, "C++ for Engineers ", *Arnold, New York - Toronto, 1996* .

[29] "A Brief History of the Green Project", *Sun Microsystems, http:/oday.java.net/jag/old/green/, 1997* .

[30] Mark Wutka, "Using Java 2 Enterprise Edition", *Que, Indiana, USA, 2001* .

[31] Various, "GNU Make", *Free Software Foundation, http://www.gnu.org/software/make/, 2004* .

[32] Various, "Apache ANT Website", *Apache Foundation, http://ant.apache.org/, 2005* .

[33] Various, "The NetBeans IDE Website", *NetBeans / Sun Microsystems, http://www.netbeans.org/, 2005* .

[34] Ramez Elmasri, "Fundamentals of Database Systems", *Cummings, USA, 1994* .

[35] Various, "Dublin Core Metadata Initiative (DCMI)", *DCMI, http://www.dublincore.org/, 2005* .

[36] Erik T. Ray, "Learning XML", *O'Reilly, Sebastopol, USA, 2001* .

[37] Various, "W3 Schools", *Refsnes Data, http://www.w3schools.com/, 1999-2005* .

[38] Various, "Java API for XML Processing (JAXP)", *Sun Microsystems, http://java.sun.com/xml/jaxp/, 2005* .

[39] Various, "Databases at CERN", *CERN, http://wwwinfo.cern.ch/db/, 2002* .

[40] R.G.G. Cattell, "Object Data Management ", *Addison-Wesley, Reading, MA, 1991* .

[41] "International Organization for Standardization", *ISO, http://www.iso.org/, 2006* .

[42] Various, "JDBC Technology", *Sun Microsystems, http://java.sun.com/products/jdbc/, 2005* .

[43] Various, "Data Access and Storage Developer Center", *Microsoft, http://www.microsoft.com/data/, 2005* .

[44] Various, "unixODBC", *Easysoft, http://www.unixodbc.org/, 2005* .

[45] Various, "Apache Xindice", *Apache Foundation, http://xml.apache.org/xindice/, 2005* .

[46] Various, "Building Scalable, High Performance Cluster and Grid Networks", *Force 10 Networks Inc., CA, USA , 2005* .

[47] Christopher R. Hertel, "Implementing CIFS (The Common Internet File System)", *Prentice-Hall PTR, http://ubiqx.org/cifs/, 1999-2004* .

[48] "NAS and SAN Technology overview", *Zerowait corp., http://www.nas-san.com, 2000-2003* .

[49] Various, "The Oscar Project", *http://oscar.sourceforge.net, 2005* .

[50] Various, "Rocks Cluster Distribution", *http://www.rocksclusters.org/, July 2005* .

[51] Paul Baran, "Publications in the 'On Distributed Communications' Series ", *RAND, http://www.rand.org/publications/RM/baran.list.html, 1964* .

[52] N Freed, N Borenstein, "Multipurpose Internet Mail Extensions, Part One: Format of Internet Message Bodies", *IETF, Network Working Group, http://www.ietf.org/rfc/rfc2045.txt, 1996* .

[53] Various, "Apache HTTP Server Project", *Apache Software Foundation, http://httpd.apache.org/, 2005* .

[54] Wolfgang Hoschek, "The Web Service Discovery Architecture", *IEEE Computer Society Press, 2002* .

[55] Various, "EU Datagrid Project", *CERN, http://eu-datagrid.web.cern.ch/eu-datagrid/, 2004* .

[56] Various, "SDSC Storage Resource Broker", *San Diego Supercomputer Center, http://www.sdsc.edu/srb/, 2005* .

[57] Various, "OGSA-DAI Website", *National E-Science centre, http://www.ogsadai.org.uk/, 2005* .

[58] "MySQL website", *MySQL AB, http://www.mysql.com/, 1995-2005* .

[59] Various, "Apache Jakarta Tomcat Webpage", *Apache Foundation, http://jakarta.apache.org/tomcat, 1999-2005* .

[60] A. Moreton, G.D Patel, T.J.V Bowcock, "GMAP - Grid Aware Monte-Carlo Array Processor", *University of Liverpool, Liverpool, 2000* .

[61] Y. Rekhter et al., "RFC 1597 Address Allocation for Private Internets", *Network Working Group, http://www.faqs.org/rfcs/rfc1597.html, March 1994* .

[62] Various, "GNU GRUB, GRand Unified Bootlander", *GNU, http://www.gnu.org/software/grub/, 2005* .

[63] K Garner, "LILO - The LInux LOader", *University of Illinois, http://www.acm.uiuc.edu/workshops/linux_install/lilo.html, 2005* .

[64] Chris Fischer, "The LoadLin+Win95/98/ME mini-HOWTO", *The Linux Documentation Project, http://www.tldp.org/HOWTO/Loadlin+Win95-98-ME.html, February 2001* .

[65] Anthony Lissot, "Linux Partition HOWTO", *The Linux Documentation Project, http://www.tldp.org/HOWTO/partition/, 2004* .

[66] Various, "Apache Tomcat Webpages, Jakarta Project", *Apache Foundation, http://jakarta.apache.org/tomcat/, 2005* .

[67] Various, "Oracle Application Server", *Oracle Corporation, http://www.oracle.com/appserver/index.html, 2005* .

[68] Jukka "Yucca" Korpela, "Methods GET and POST in HTML forms - what's the difference?", *Jukka "Yucca" Korpela, http://www.cs.tut.fi/ jkorpela/forms/methods.html, 2003-09-28* .

[69] James W Cooper, "The Design Patterns Java Companion", *Addison-Wesley, 1998* .

[70] Various, "GridPP UK Computing for Particle Physics", *GridPP, http://www.gridpp.ac.uk/, 2005* .

[71] Various, "GridSite", *GridPP/PPARC, http://www.gridpp.ac.uk/gridsite/, 2005* .

[72] Slava Pestov, "JEdit Website", *http://www.jedit.org/, 2005* .

[73] Ingyu Kang, "Crimson Editor Website", *http://www.crimsoneditor.com, 1993-2003* .

[74] Various, "Java 2 Platform, Standard Edition", *Sun Microsystems, http://java.sun.com/j2se/, 1994-2005* .

[75] Mark Matthews, "MM MySQL Drivers", *SourceForge, http://mmmysql.sourceforge.net, 2002* .

[76] James Clark, "Relax NG Website", *James Clark, http://www.relaxng.org/, September 2003* .

[77] Various, "How Stuff Works ", *HowStuffWorks, inc., http://www.howstuffworks.com, 1998-2005* .

[78] Kurt Wall, Mark Watson and Mark Whitis et al., "Linux Programming", *SAMS, Indiana USA, 1999* .

[79] Various, "Microsoft Windows Family", *Microsoft, http://www.microsoft.com/windows/, 2005* .

[80] Various, "Beowulf Website", *http://www.beowulf.org/, 2005* .

[81] Various, "MPI: A Message-Passing Interface Standard (Version 1.1)", *University of Tennessee, Knoxville, http://mpi-forum.org/docs/mpi-11-html/mpi-report.html, 1995* .

[82] Various, "MPI-2: Extensions to the Message-Passing Interface", *University of Tennessee, Knoxville, http://mpi-forum.org/docs/mpi-20-html/mpi2-report.html, 1997* .

[83] Various, "Parallel Virtual Machine ", *Oak Ridge National Laboratory, http://www.csm.ornl.gov/pvm/pvm_home.html/, 2005* .

[84] Various, "DCE Portal", *The Open Group, http://www.opengroup.org/dce, 1995/2005* .

[85] Various, "CORBA FAQ", *Object Management Group, http://www.omg.org/gettingstarted/corbafaq.htm, 1997-2005* .

[86] Markus Horstmann and Mary Kirtland, "DCOM Archictecture", *Microsoft Developers Network, http://msdn.microsoft.com/library/en-us/dndcom/html/msdn_dcomarch.asp, 1997* .

[87] William Grosso, "Java RMI ", *O'Reilly, Sebastopol , 2002* .

[88] Gopalan Suresh Raj, "A Detailed Comparison of CORBA, DCOM and RMI", *http://my.execpc.com/ gopalan/misc/compare.html, 1998* .

[89] Various, "Wikipedia", *Online encyclopedia, http://www.wikipedia.org/, 2005* .

[90] Danny Ayers, Hans Bergsten, Michael Bogovich, Jason Diamond, Matthew Ferris, Marc Fleury, Ari Halberstadt, Paul Houle, Piroz Mohseni, Andrew Patzer, Ron Phillips, Sing Li, Krishna Vedati, Mark Wilcox and Stefen Zeiger, "Professional Java Server Programming", *Wrox, Birmingham, UK, 1999* .

[91] Various, "W3C's pages on Markup Languages", *W3C, http://www.w3c.org/MarkUp/, 2005* .

[92] Jeffrey Richter, "Applied Microsoft .NET Framework Programming", *Microsoft Press, Redmond, USA, 2002* .

[93] Scott Short, "Building XML Web Services for the Microsoft .NET Platform", *Microsoft Press, Redmond, USA, 2002* .

[94] "Jini Network Technology", *Sun Microsystems, http://www.sun.com/software/jini, 2005* .

[95] "Jini Community ", *CollabNet Inc, Sun Microsystems, http://www.jini.org/, 2005* .

[96] Perdita Stevens, Rob Pooley, "Using UML. Software Engineering with Objects and Components", *Addison-Wesley, Harlow, Essex, England, 2000* .

[97] Various, "Proceedings of the Third IEEE META-DATA Conference", *IEEE, http://www.computer.org/proceeding/meta/1999/, April 1999* .

[98] Various, "DocBook ", *O'Reilly, http://www.docbook.org/, 2005* .

[99] Various, "Oasis Open Website", *Oasis Open, http://www.oasisopen.org/, 2005* .

[100] Various, "Scalable Vector Graphics (SVG) 1.1 Specification", *W3C, http://www.w3.org/TR/SVG/, January 2003* .

[101] Various, "WAP/WML Tutorial", *W3 Schools, http://www.w3schools.org/*, .

[102] Jason Hunter, "JDOM Website", *JDOM Project, http://www.jdom.org/, 2005* .