UNIVERSITY OF Southampton

University of Southampton Research Repository
ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

http://eprints.soton.ac.uk

University
of Southampton

Encouraging collaboration through a new data management approach.

Steven Johnston[1]

Computational Engineering and Design Group
School of Engineering Sciences
University of Southampton
United Kingdom

Supervisors: Prof. Simon J. Cox[2] , Dr. Hans Fangohr[3]

Date: August 2006

[1]sjj698@zepler.org
{[2]sjc , [3]fangohr}@soton.ac.uk

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ENGINEERING SCIENCES

Doctor of Philosophy

ENCOURAGING COLLABORATION THROUGH A NEW DATA MANAGEMENT APPROACH

Steven James Johnston

The ability to store large volumes of data is increasing faster than processing power. Some existing data management methods often result in data loss, inaccessibility or repetition of simulations. We propose a framework which promotes collaboration and simplifies data management.

In particular we have demonstrated the proposed framework in the scenario of handling large scale data generated from biomolecular simulations in a multi-institutional global collaboration. The framework has extended the ability of the Python problem solving environment to manage data files and metadata associated with simulations. We provide a transparent and seamless environment for user submitted code to analyse and post-process data stored in the framework.

Based on this scenario we have further enhanced and extended the framework to deal with the more generic case of enabling any existing data file to be post processed from any .NET enabled programming language.

Contents

List of Figures

List of Tables

List of Examples

Authors declaration

I, Steven Johnston declare that this report entitled *Encouraging collaboration through a new data management approach* and the work presented in it, are my own.

I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published in:

  Arinaminpathy Y, Beckstein O, Biggin P, Bond P, Domene C, Pang A and Sansom M. Large scale biomolecular simulations: Current status and future prospects. *Proceedings of UK e-Science All Hands Meeting* (2003).

  Boardman RP, Johnston S, Essex JW, Ng M, Fangohr H, Tai K and Sansom MSP. BioSimGrid on the desktop (2006). In preparation for submission.

  Jiao Z, Wason J, Molinari M, Johnston S and Cox S. Integrating data management into engineering applications,. *Proceedings of UK e-Science All Hands Meeting, Nottingham, UK*, pages 687 – 694 (2003).

  Johnston S. Storage resource broker (SRB), large scientific data and Python. *Europython 2005* (2005).

  Johnston S, Boardman R, Fangohr H and Cox S. Managing large volumes of distributed scientific data. *Journal of Grid Computing* (2006a). (Submitted).

  Johnston S, Boardman RP, Ng M, Essex JW, Fangohr H, Tai K and Sansom MSP. BioSimGrid: Infrastructure, performance and applications (2006b). In preparation for submission.

Murdock S, Muan Hong Ng, Johnston S, Fangohr H and Essex J. Comparing the performance of a database, specific binary files and netCDF for data retrieval (2004). [Online; accessed 10-December-2005], `www.biosimgrid.org`.

Murdock SE, Tai K, Ng M, Johnston S, Wu B, Fangohr H, Laughton CA, Essex JW and Sansom MSP. Quality assurance for biomolecular simulations. *Journal of Chemical Theory and Computation* (2006). Sumbitted.

Ng M, Johnston S, Murdock S, Wu B, Tai K, Fangohr H, Cox S, Essex JW, Sansom M and Jeffreys P. Efficient data storage and analysis for generic biomolecular simulation data. *In Proceedings of UK e-Science All Hands Meeting 2004*, pages 443–450 (2004).

Ng M, Johnston S, Wu B, Murdock S, Tai K, Fangohr H, Cox SJ, Essex JW, Sansom MSP and Jeffreys P. BioSimGrid: Grid-enabled biomolecular simulation data storage and analysis. *Future Generation Computer Systems*, **22**, 657–664 (2006).

Tai K, Baaden M, Murdock S, Wu B, Ng M, Johnston S, Boardman R, Fangohr H, Cox K, Essex JW and Sansom MSP. Three hydrolases and a transferase: comparative analysis of active-site dynamics via the BioSimGrid database. *Journal of Molecular Graphics and Modelling* (2006).

Tai K, Murdock S, Wu B, Ng M, Johnston S, Fangohr H, Cox SJ, Jeffreys P, Essex JW and Sansom MSP. BioSimGrid: Towards a worldwide repository for biomolecular simulations. *Organic & Biomolecular Chemistry*, **2**, 3219–3221 (2004).

Woods CJ, Ng M, Johnston S, Murdock SE, Wu B, Tai K, Fangohr H, Jeffreys P, Cox S, Frey JG, Sansom MSP and Essex JW. Grid computing and biomolecular simulation. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, **363**(1833) (2005).

Wu B, Dovey M, Ng M, Tai K, Murdock S, Fangohr H, Johnston S, Jeffreys P, Cox S, Essex J and Sansom MSP. A web / Grid portal implementation of BioSimGrid: A biomolecular simulation database. *Journal of Digital Information Management*, **2**(2), 74–78 (2004a).

Wu B, Dovey M, Tai K, Ng M, Stuart, Murdock, Fangohr H, Johnston S, Jeffreys P, Cox S, Essex JW and Sansom MS. Security and BioSimGrid: A biomolecular simulation database. *Proceedings of Workshop on Grid Security Practice and Experience* (2004b). Published as: University of York, Department of Computer Science Technical Report YCS-2004-380.

Wu B, Tai K, Murdock S, Ng M, Johnston S, Fangohr H, Jeffreys P, Cox S, Essex J and Sansom MS. BioSimGrid: A distributed database for biomolecular simulations. *Proceedings of UK e-Science All Hands Meeting 2003*, pages 412–419 (2003).

Wu B, Tai K, Ng M, Johnston S, Murdock S, Fangohr H, Sansom MSP, Essex J, Jeffreys P and Cox S. Towards a Grid-enabled biomolecular simulation database. *In Proceedings of UK e-Science All Hands Meeting 2005*, pages 577–580 (2005).

The BioSimGrid project is a collaborative project involving the work of others of which I developed and implemented the flatfile storage layer, consisting of controlling software and data structures. The work involving the analysis tools and metadata schema are the work of others.
The author recognises the work of the following:

Simon Cox and Hans Fangohr for suggesting some data management mechanisms and testing criteria, in particular, using flatfiles and object serialisation to manage data.

Name :

Signature :

Date :

Acknowledgements

I would like to thank Hans Fangohr and Simon Cox for their supervision and support throughout my research. Many thanks to Richard Boardman for his technical expertise, proofreading and LaTeX templates, Ian Hartney for the eTex project and the valued members of the BioSimGrid team.

A special thanks to those who helped with proof reading and corrections, along with the moral support required to finish.

Copyrights and trademarks

- Apple® , Mac OS® and Macintosh® are registered trademarks of Apple Computer, Inc.
- CORBA® is a registered trademark of the Object Management Group (OMG).
- HTML™, XML™ and W3C® are trademarks or registered trademarks of W3C® , World Wide Web Consortium.
- IBM® is a registered trademark of IBM in the United States of America and/or other countries.
- Java™ is a trademark of Sun Microsystems, Inc.
- J2EE™ is a trademark of Sun Microsystems, Inc.
- Linux® is a registered trademark of Linus Torvalds.
- Microsoft® Software, Windows® Operating system and Microsoft .NET™ are either registered trademarks or trademarks of the Microsoft Corporation.
- Oracle® is a registered trademark of the Oracle Corporation.
- Python™ is a trademark of the Python Software Foundation.

List of Acronyms

ACID	Atomicity Consistency Isolation and Durability
ADAM	Active Directory Application Mode
AFS	Andrew file system
API	Application Programming Interface
BFS	Be File System
BLOB	Binary Large Object
CERN	Conseil Européen pour la Recherche Nucléaire
CIFS	Common Internet File System
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLOB	Character Large Object
CLR	Common Language Runtime
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CTP	Community Technology Preview
CVS	Concurrent Versions System
DBA	Database Administrator
DBMS	Database Management System
DCOM	Distributed Component Object Model
DDL	Data Definition Language
DFS	Distributed File System
DIME	Direct Internet Message Encapsulation
DLL	Dynamically Linked Library
DNA	Deoxyribonucleic acid
DOM	Document Object Model
DQL	Database Query Language
DS	Digital Signal
DSL	Digital Subscriber Line
EB	Exabyte (10^{18} bytes)
EiB	Exbibyte (2^{60} bytes)

FAT	File Allocation Table
FODB	File Object Database
FOM	File Object Method
GB	Gigabyte (10^9 bytes)
GEODISE	Grid Enabled Optimisation and Design Search for Engineering
GiB	Gibibyte (2^{30} bytes)
GID	Globally unique Identifier
GPL	GNU General Public License
GUI	Graphical User Interface
Gnome VFS	Gnome Virtual File System
HDD	Hard Disk Drive
HFS	Hierarchical File System
HPC	High Performance Computing
HTTP	HyperText Transfer Protocol
ID	Identifier
iFS	Internet File System
IPSec	Internet Protocol Security
IRQL	Information Retrieval Query Language
ISBN	International Standard Book Number
JACS	Journal of the American Chemical Society
JAX-WS	Java API for XML Web Services
J2EE	Java 2 Platform Enterprise Edition
JIT	Just-in-Time
JVM	Java Virtual Machine
KB	Kilobyte (10^3 bytes)
KiB	Kibibyte (2^{10} bytes)
LAN	Local Area Network
LOB	Large Object
LUW	Logical Unit of Work
MB	Megabyte (10^6 bytes)
MCAT	Metadata Catalog
MD	Molecular Dynamics
MiB	Mebibyte (2^{20} bytes)
MIME	Multipurpose Internet Mail Extensions
MMR	Multi-Master Replication

MTOM Message Transmission Optimisation Mechanism

NFS Network File System
NMR Nuclear Magnetic Resonance
NTFS New Technology File System

OASIS Organisation for the Advancement of Structured Information Standards
OC Optical Carrier
ODBMS Object Orientated Database Management Systems
OFS Object File System
OLE Object Linking and Embedding
OO Object Oriented
OS Operating System
OSS Open Source Software

PB Petabyte (10^{15} bytes)
PCCP Physical Chemistry Chemical Physics
PDB Protein Data Bank
PiB Pebibyte (2^{50} bytes)
PL/SQL Procedural Language/SQL
PNNL Pacific Northwest National Laboratory
PSE Problem Solving Environment

QoS Quality of Service

RDB Relational Database
RDBMS Relational Database Management System
RDF Resource Description Framework
RMI Remote Method Invocation

SDSC San Diego Supercomputing Center
SQL Structured English Query Language
SETI Search for Extraterrestrial Intelligence
SIS Single Instance Storage
SMR Single-Master Replication
SN Strong Name
SOA Service-Oriented Architecture
SOAP Simple Object Access Protocol
SPF Storage and Processing Framework
SQL Structured Query Language
SQL PL SQL Procedural Language
SRB Storage Resource Broker
SSCLI Shared Source Common Language Infrastructure

SSL	Secure Sockets Layer
SwA	SOAP with Attachments
TB	Terabyte (10^{12} bytes)
TiB	Tebibyte (2^{40} bytes)
UCSD	University of California, San Diego
UDDI	Universal Discovery, Description, Integration
URI	Uniform Resource Indicator
WinFS	Windows Future Storage
WSDL	Web Services Description Language
WSE	Web Services Enhancements
WS-RM	WS-ReliableMessaging
WWW	World Wide Web
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XOP	XML-binary Optimised Packaging
XPath	XML Path Language
XQuery	XML Query

Chapter 1

Introduction

Science and engineering are generating vast quantities of data. Each year these volumes increase as advances are made in computer hardware and software. The problems associated with managing these volumes of data are continually evolving to cope with the ever growing data volumes.

Unlike industries such as banking or retail there are few purpose built infrastructures or applications specifically designed for scientific or engineering data. This is partly because of the complexity and volume of the data, and partly because of the specific applications. Although there exists a plethora of general data management tools which are capable of dealing with general data, such as databases. They are unsuitable for scientific users as they often require a good understanding of complex data management issues.

From this we can see that there is a need to bridge the gap between complex data management applications and non-technical scientific users.

The following data properties and issues are important for the understanding of this work:

- Data and information

  Data and information are often incorrectly taken as synonyms, but their meanings are different. Data is the representation of information, for example the number *two* provides the same information regardless of the representation, *i.e.* 2 , II , two.

  Data may appear unorganised and without meaning and it is only when interpreted or organised that it becomes information. Information passed on without understanding becomes data, *i.e.* one person's information may be another person's data (*Date*, 2000a).

- Large data volumes

  Scientific datasets are continuously increasing in size, partly due to progresses in hardware and partly due to increased accessibility of High Performance Computing (HPC) resources. An entry level desktop computer can expect to

store data in the 100's of Gigabytes (GB). Today most scientific data sets can expect to require 100s of GB of data and some scientific datasets reach the 10s of Terabytes (TB). There are a few which are expected to reach multiple Petabyte (PB) of data. For example at Conseil Européen pour la Recherche Nucléaire (CERN) the Large Hadron Collider is expected to produce over a PB of High Energy Physics data a year (*Harbottle et al.*, 2003), amounting to over 100 PB of data in its operational lifetime.

- Collaborations

  As larger and larger problem spaces are investigated, the greater the importance of collaborations. Collaborations extend between individuals as well as institutions and across disciplines. They enable large complex problems to be approached and can help reduce repetition of work.

- Data generation lifecycle

  The scientific data lifecycle starts with the creation of data, usually through simulations or experiments. This data is then stored for future analysis. The data life cycle raises issues about how long data should be stored and who should have access.

In this thesis we investigate the existing technologies available for data management and discuss why existing technologies are not readily adopted. With the rapid increase in storage availability we investigate if there is a need to introduce more efficient data management methods. Processing power is increasing at a slower rate than network bandwidth and storage capacity thus the emphasis is moving from computation of data to the storage of data. Although storage is becoming abundant, there is a need to efficiently utilise the processing power required to generate, retrieve and store the data. For example, storing data is useless unless you can retrieve it promptly. The processing required to store and retrieve the data needs to be less that the processing power required to generate the data. We investigate the current non-technical scientific users data workflow and investigate how future data collaborations can be encouraged.

Data varies in complexity and therefore the techniques required to manage the data differ. We investigate the issues which complicate data management and demonstrate how these relate to the existing data management technologies. We intend to use this investigation to highlight limitations in the existing workflow and show how potential collaboration opportunities are hindered.

We use the findings to propose a new data management method, called the File Object Method (FOM). This aims to introduce complex database features to the average scientist. We propose a method to manage data without the user having to learn complex new technologies. The proposed method can enhance the existing

users' workflow and we outline the key features which the model provides end users.

After introducing the implementation-independent method we consider a prototype implementation of the new data management method, called the File Object Database (FODB). We discuss how the model can work on sample data and show how a specific implementation can implement the features discussed in the data management method. The prototype provides a proof of concept highlighting the advantages and disadvantages of the proposed method.

We use the proposed method and findings from the prototype to implement a feature limited version implementation. This version is used to manage large volumes of scientific data in an existing scientific project called BioSimGrid.

Using the findings, we suggest how the data management method can be improved. These changes are then used to produce a final prototype of the data management method called the Storage and Processing Framework (SPF). We compare and contrast the SPF with the FODB and the BioSimGrid project, showing how they meet the objectives of this thesis.

The objectives of this thesis are as follows:

- Data management framework —To provide a framework capable of managing large volumes of scientific data.
- Resource utilisation management — To reduce the loss or inaccessibility of computational data which results in the repetition of expensive computations.
- Data organisation — To organise data such that users can easily identify and analyse the contents.
- Locate data — To enable users to efficiently locate data within the framework.
- Commodity systems — To utilise commodity systems and take advantage of the power-to-price benefits and to minimise the requirements on the end users.
- Promote collaborations — To enable the sharing of software algorithms and code specific to the data sets, between users where appropriate.
- Generic data — To produce a data domain independent framework for managing data.
- Generic resources — To provide a platform and hardware independent framework for managing data.
- Utilise open standards — To produce a framework which both utilises and supports open standards wherever possible.

This thesis is structured as follows:

- Chapter 1 — In this current chapter we introduce the existing data management issues. We propose investigating the existing issues and outline a series of objectives for this work.
- Chapter 2 — This chapter provides an outline of the issues faced when managing large volumes of scientific data. We look at the data workflow and provide an overview of the existing data management technologies.
- Chapter 3 — We introduce a generic framework concept to address the management of data with minimal interference to the existing users workflow.
- Chapter 4 — In this chapter we take the generic framework discussed in chapter 3 and demonstrate its features and capabilities using an example implementation.
- Chapter 5 — In this chapter we show how the framework discussed in chapter 3 is used to manage a large volume of scientific data. The framework is demonstrated as part of a practical application to form a globally accessible repository used by computational scientists.
- Chapter 6 — This chapter provides a critical analysis of the framework implementations and recommends improvements to the proposed framework.
- Chapter 7 — In this chapter we use the recommendations provided in chapter 6 to produce an improved implementation of the framework.
- Chapter 8 — In this chapter we compare and contrast the various framework implementations with the initial generic framework description. We show how the objectives outlined in chapter 1 met.
- Chapter 9 — This chapter we summarise all the findings of this work.

Chapter 2

Data and data management

2.1 Introduction

In this chapter we look into the current life cycle of scientific data, we identify limitations in the current model and show how collaborations could avoid repeating simulations.

There are many existing technologies aimed at managing and maintaining data but these are often misused; the reasons for this are discussed below.

We discuss the different technologies capable of dealing with data and outline their basic capabilities. The relevant infrastructure and middleware components are described including some of the most recent technologies.

The chapter closes with a summary of the limitations of the existing technologies. We provide an indication of the improvements that are possible. It concludes with an emphasis on bringing database capabilities to the scientist and seamlessly integrating them into their existing workflows. The overall aim is to leverage existing technologies to assist scientists without increasing the overall complexity of the scientific workflow.

2.2 Accumulation of scientific data

In day to day conversation it is common to hear people discussing data volumes, for example the number of emails received, music player capacity or even the number of pages in a book. These measurements are often open to interpretation and introduce ambiguity. The international system of units, universally abbreviated SI from the French Le Système International d'Unités, are also ambiguous when related to computing.

Storage data capacity terms have popular use meanings which are different. For example a Kilobyte (KB) of data is 2^{10} bytes, *i.e.* 1024 characters, but is often referred to as 10^3 bytes of data. In this example the difference is minimal and often ignored. As the data volumes get very large, for example an Exabyte (EB) is 2^{60} bytes and

Table 2.1: Shows the SI and IEC 60027-2 standard units utilised in this work to denote data volumes.

SI units	Bytes	IEC 60027-2 units	Bytes
Kilobyte (KB)	10^3	Kibibyte (KiB)	2^{10}
Megabyte (MB)	10^6	Mebibyte (MiB)	2^{20}
Gigabyte (GB)	10^9	Gibibyte (GiB)	2^{30}
Terabyte (TB)	10^{12}	Tebibyte (TiB)	2^{40}
Petabyte (PB)	10^{15}	Pebibyte (PiB)	2^{50}
Exabyte (EB)	10^{18}	Exbibyte (EiB)	2^{60}

is often referred to as 10^{18} bytes. The difference between the two is vast enough to make the difference important. In this work we utilise the IEC 60027-2 standard to denote data volumes *i.e.* a KB is 2^{10} bytes and written as Kibibyte (KiB), as shown in table 2.1.

Moore's Law (*Moore*, 1965) states that the number of transistors on a chip doubles every 18 months and based on this it is often quoted that computing power doubles every 18 months. Although not exactly correct as it does not take into account the processor clock speed or the software algorithms it is, nevertheless a good indicator of the future performance of hardware. It is generally accepted that processing power doubles every 18 – 24 months and data storage density every 12 – 18 months (*Stix*, 2001). The result is that the cost per Gibibyte (GiB) (*IEC*, 2005) to the end user is falling which indicates that the ratio of processing power to data storage will decrease, as shown in figure 2.1. This opens up new opportunities to consider more efficient methods of data management. This is already becoming apparent for the average home user who is unable to manage the location and content of files on a single local machine. The cost of storage has reduced so that it is easily possible to build a desktop machine with a Tebibyte (TiB) (*IEC*, 2005). Having this volume of storage easily accessible gives rise to potential organisational issues. For example, one TiB of holiday photos taken at high resolution is approximately a quarter of a million photos. Managing these many photos easily would require a photo management application. The same applies to scientific data although no general scientific data management application exists.

Figure 2.1 shows that in the future a unit of processing power will be responsible for managing a larger volume of storage space. This indicates that there is a need to improve and rethink some of the current techniques used to manage data.

IBM (*IBM*, 2005) is currently working with CERN (*CERN*, 2005) on a system capable of dealing with a Pebibyte (PiB) (a million GiB) of data (*Harbottle et al.*, 2003), with the aim to provide scientists with access to their data, local or remote, using any operating system. There are various solutions available today to organise large amounts of data, ranging from file systems to databases. Each method requires a certain amount of setup costs and has both strengths and weaknesses.

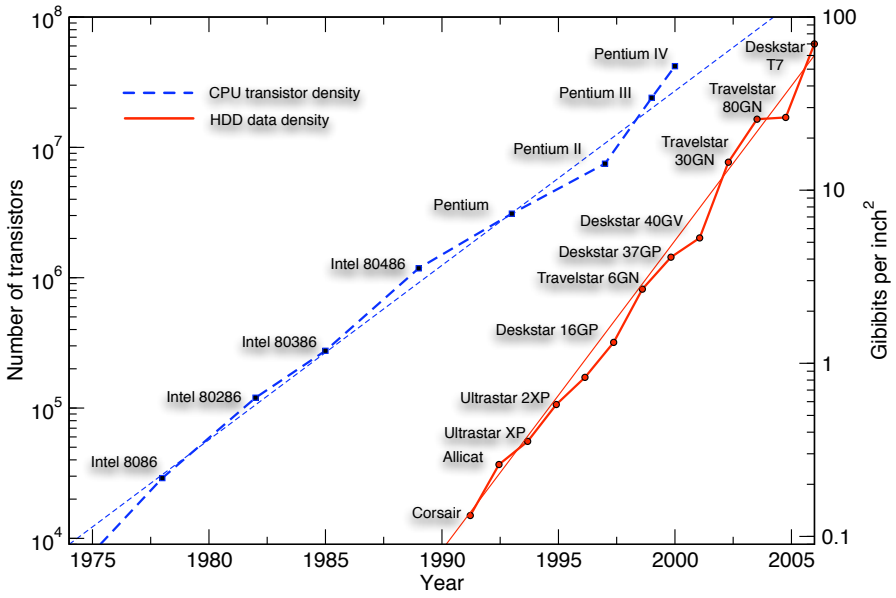

Figure 2.1: Shows how the transistors density and data storage density has increased exponentially in the past. Although this indicates that processing power and storage capabilities are increasing exponentially, they are not increasing at the same rate. HDD capacity doubles every 12 to 18 months while processing power increases every 24 to 36 months. (*Tuomi*, 2002, *Grochowski*, 2003, *Thompson and Best*, 2000)

Ideally it would make sense to have all results in a centrally located database which is accessible by all the scientists in that field. Although the volume of the data is a potential problem, organising the data and encouraging the scientists to use a database is a key issue. Ignoring the privacy and copyright drawbacks of having a shared database, there is potential to save time by not repeating experiments. This would require a database that is easy to setup and populate, ideally without the user having to change any coding practices.

However with all the organisational technology available to scientists, it is still common to find experiment results being stored in files in a proprietary format. Often experiment results are stored locally in flatfiles which are then processed using command line utilities, combined with custom programs. The net result is that experimental data can only be accessed by users with access to the data files and an understanding of the contents of each file. Often the data is large, and storing many sets of results can consume large volumes of space which may not be available, so the data is deleted. The overall result is that lots of data becomes inaccessible or lost due to the storage practices used, resulting in the same experiment having to be run multiple times.

One way to encourage users to make data accessible to others is to restrict or

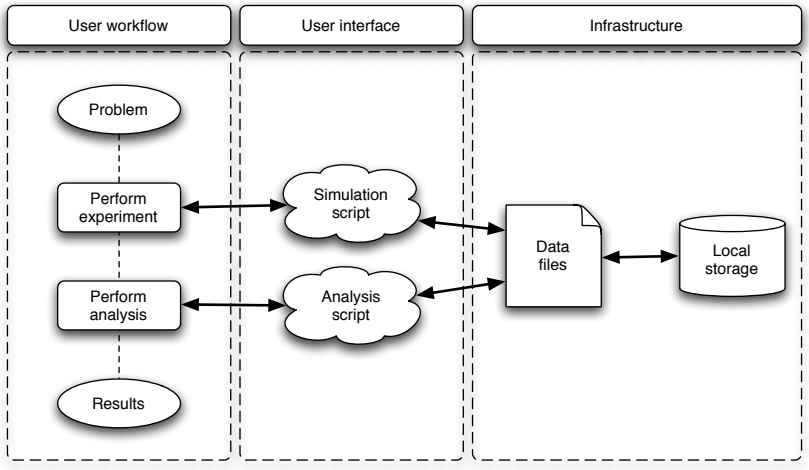

Figure 2.2: Shows the general life cycle of scientific data looking at the users workflow and interfaces as well as the infrastructure used.

generalise the format in which the data is stored, but this is difficult to impose on users. It is not possible to convince every scientific user to comply to a single data standard, as no single standard is sufficient. For example if we look at Extensible Markup Language (XML) (*W3C*, 2004a), it is a general data standard which forces users to format data in a manner which can be read by any XML parser, however it severely bloats the data and is impracticable for a large data set.

This section looks into the complexity of data and summarises the capability of existing technology to manage different kinds of data.

2.2.1 Scientific data life cycle

Figure 2.2is divided into workflow, interface and infrastructure sections. The workflow section shows the tasks a user can perform. These are linked to the interface section, which shows how the user would perform a task in the workflow. For example this section will include any tools which the user uses and any custom code created. The infrastructure section shows the physical management of data, including the data files and hardware. The system is described in these three sections to show that each section is independent.

In figure 2.2 the user begins by identifying a problem and then performs a simulation using suitable applications or scripts. The data output from these simulations is then analysed to address the initial problem.

The data files are stored on a local storage medium which is inaccessible to the community at large. The simulation and analysis scripts are often specific to the current data and are not subject to any formal storage or quality controls.

2.2.2 Storage versus repeat simulations

The need to store and share data within a community is only applicable for certain types of data. Where the time taken to store and retrieve data is greater than the time taken to recompute the data, recompute is favourable. The cost of Hard Disk Drive (HDD) storage and processing time also has to be considered. A case where data is more efficiently stored rather than recomputed can be shown as:

$$C_{\text{hardware}} + C(T_{\text{storage}}) + C(T_{\text{retrieval}}) + C_{\text{storage}} < C(T'_{\text{compute}}) + C'_{\text{hardware}} \quad (2.1)$$

where C_{hardware} is the cost of hardware to compute the data, T_{storage} is the time taken to store the data, $T_{\text{retrieval}}$ is the time taken to retrieve the data, C_{storage} is the cost of storing the data, T'_{compute} is the time taken to recompute the data and C'_{hardware} is the cost of hardware to compute the data. $C(T_{\text{storage}})$ and $C(T_{\text{retrieval}})$ include the cost of storage and retrieval in terms of network cost and transmission time.

As the cost of hardware shown in equation 2.1 varies over time and the hardware used may vary depending on the urgency of a job. The values for C_{hardware} and C'_{hardware} may not be the same.

Often programmers recalculate data where memory or bandwidth is constrained (*Kandemir et al.*, 2005), but as shown in figure 2.1, future processing power is expected to become the main constraint.

In the case of scientific simulations it is often the case that the simulation is processor intensive and although the data is large it is beneficial to store the data.

In scientific research which involves intensive simulations, it is often beneficial to store the data for reuse even if this requires large storage space.

Providing a mechanism to show how results are computed and providing access to the data enables a comparison of different analysis techniques and algorithms. This will further provide opportunities for peers to recalculate and confirm findings without rerunning simulations.

2.2.3 Limitations

Looking at technology trends and the existing scientific data life cycle the following limitations are observed.

- Collaborations are very difficult when data is stored locally. This is divided into two problem areas, finding the data and accessing the data. Collaborations cannot form if external parties are unaware of what data is stored. Secondly when data is stored locally, access permissions and differences in data formats are often enough to curb potential collaborative opportunities.

- Backing up large volumes of data is often expensive and not feasible. A backup and restore can take longer than the data can acceptably be offline. The best form of backup is data redundancy, either multiple copies per site or across distributed sites. This ensures that the data is always available should a site or copy become corrupted. The current method of local storage does not easily adapt to convenient backup strategies.

- Storing data locally is no longer an accepted strategy to store valuable data, even if it is replicated many times. HDD failures are common, making a single instance of data very vulnerable. Even if the file system has some hardware redundancy (*Vadala*, 2002a) as disks from the same batch tend to fail at the same time. Even multiple copies on a single site are unacceptable as this is exposed to site risks like power surges, fire and theft.

- Once the data is calculated it is then analysed using scripts or programs. Users perform the analysis using scripts which they have written or acquired from colleagues. It is not possible to assess the reliability of results published. In order to validate results scientists have to either ask for the analysis scripts or write their own in an attempt to reproduce the results. This leads to repetition of work and indicates the need for a mechanism for exchanging analysis scripts.

2.2.4 Data transmission costs

The cost of moving data is often overlooked. When data volumes are small the cost can safely be ignored. As data volumes become increasingly large the transmission of data becomes a limitation both in terms of speed and cost. In table 2.2 we look at some of the various network connections available. The modem or narrow band connection is so slow that the rental charge becomes a dominant factor. One of the most economical solutions is Digital Subscriber Line (DSL) which is easily available and comparably inexpensive. The 3,600 hours taken to transmit one TiB of data makes DSL unsuitable.

We then look at the cost of a Trunk or Digital Signal (DS) connection, the DS–1 is one of the more expensive per TiB and still takes an unacceptable 1,311 hours to transmit one TiB. The DS–3 is the first practical connection suitable of dealing with large volumes of data. It sets the transmission costs of data at around $2,000 per TiB.

The Optical Carrier (OC) connection provides a suitable bandwidth but is expensive and is best utilised by large institutions with a constant high bandwidth demand.

Moving data is expensive and the cost of high bandwidth connections can quickly outweigh the cost of physically posting the data on a HDD. This option has a

Table 2.2: The speed, cost and time of data transmissions over various network connections. The table includes a comparison of Digital Signal (DS) and Optical Carrier (OC) connections as well as home connections like dial-up or narrow band and Digital Subscriber Line (DSL). Data shown in USD and courtesy of Jim Gray at Microsoft.

Method	Speed (Mbps)	Rent ($/month)	$/Mbps	$/TiB (sent)	Time/TiB (hours)
Narrow band	0.04	40	1,000	3,086	52,560
Home DSL	0.6	70	117	360	3,600
DS–1	1.5	1,200	800	2,469	1,440
DS–3	43	28,000	651	2,010	48
OC–3	155	49.000	315	976	14
OC–192	9,600	1,920,000	200	617	0.23
Postal system	–	–	–	50	24

high latency ($\approx$ 24 hours), but it is capable of moving multi-TiB of data in a single operation. Since the cost of moving data via a network exceeds $2,000 per TiB this is often a viable solution.

In table 2.3 we look at the time taken to transmit one TiB of data over a LAN at different speeds. Over a 100 Mbps Local Area Network (LAN) transferring one TiB takes an entire day and on a high speed 1 Gbps LAN over 2 hours. Transferring

Table 2.3: Shows the expected time taken to transmit one TiB of data across a local area network.

LAN	Time/TiB (hours)
10 Mbps	250
100 Mbps	24
1 Gbps	2.2
10 Gbps	0.21

data is expensive and requires suitable quantities of storage at either location to manage the large volumes of data. Even on a LAN the time is still significant. It is for these reasons that moving the compute closer to the data is preferable to moving the data to the compute.

2.3 Data complexity

There are many naturally occurring characteristics of data that make it difficult to manage and it is this complexity that often determines how the data is stored. An example of a complex dataset is given in figure 2.3 which shows the relationship of experimental results with publications and authors. The complexity of data is related to the number of relations between each data item; the more relations there are, the more complex the data. There are also different types of relations, one-to-

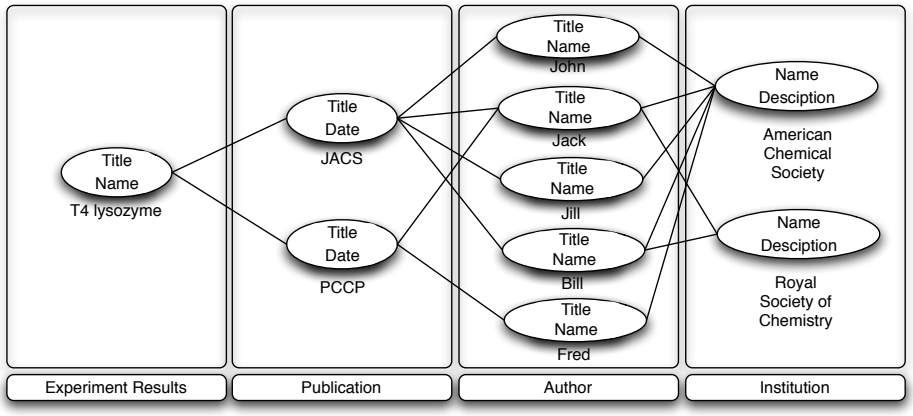

Figure 2.3: Shows a graphical representation of complex metadata. This metadata for a `T4 Lysozyme` experiment shows it has been published in two journals and some of the authors worked on both publications

one which is the simplest, one-to-many which is more complex and many-to-many which is the most complex. The relations between each data item complicates the data due to the storage methods. Each relation has to be stored and managed so that it can be traversed and retrieved.

Figure 2.3 shows how the example experimental results, `T4 Lysozyme` has been used in two publications, one in the Journal of the American Chemical Society (JACS) and the other in the Physical Chemistry Chemical Physics (PCCP) journal (*JACS*, 2004, *PCCP*, 2004). Each of these publications has multiple authors and some of the authors (`Jack, Bill`), worked on both publications. All of the authors belong to the American Chemical Society and some also belong to the Royal Society of Chemistry.

It is common to find data with complex relationships and dependencies. In order to address the issues that this poses, it is important to look at the reasons why the data management is considered complex.

2.3.1 Natural unique identifiers

Complex data often lacks natural unique identifiers. When storing data it has to be uniquely identified so that the correct data can be retrieved at a later date. The unique Identifier (ID) can take many forms:

- ID unique to the current dataset.
  For example a regional telephone number, `123456`.
- ID unique to the current system.
  For example a national telephone number, `023 123456`.
- Globally unique ID, such as ISO 3166-2 (*ISO*, 1995).
  For example an international telephone number, `44 023 123456`.

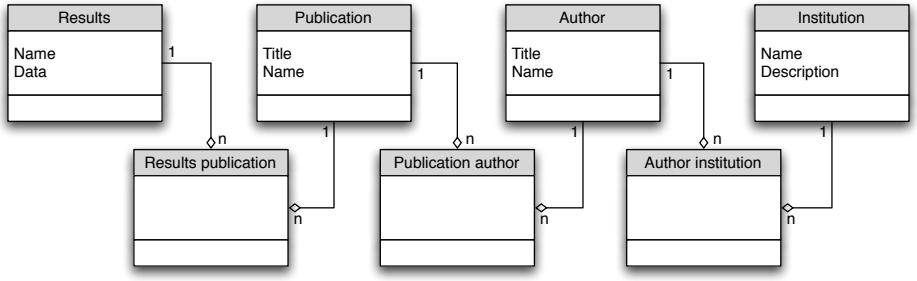

Figure 2.4: Shows how linking tables are required to represent many-to-many relations in a relational database. These linking tables have one-to-many (1:N) relationships with the parent tables resulting in a many-to-many relationship mapping.

Each of these identifiers adds complexity to the data as the uniqueness has to be carefully managed or the data will become corrupted. Complex data often identifies itself through relationships with other data and does not contain single unique IDs. This means that the data storage layer has to allocate and manage IDs for all the data that is being stored, resulting in more data needing to be stored.

Even the simple case of allocating integers to every record added can cause problems in a volatile environment. It is possible to run out of numbers that can fit into an integer field and hence ID recycling has to be introduced as well as the use of multicolumn keys known as complex composite keys (*Getz et al.*, 1994).

2.3.2 Many-to-many relationships

Complex data often contains large numbers of many-to-many relationships. It is easy to manage one-to-one and one-to-many relationships and many systems are built to deal with this. Most databases and file systems are designed with the one-to-one and one-to-many model in mind but it poses limitations for many-to-many relationships.

It is possible to model many-to-many relationships using only one-to-many relations by adding an intermediate mapper that maps the relations. This is often unacceptable as it involves more work, more data and more processing. The mapper has to be managed and the data either side of the relationship has to have unique identifiers.

Using a mapper not only adds complexity at the data level but also at the retrieval level as the accessing applications have to be able to understand these mappings and be able to reconstruct the data using the unique IDs.

Many-to-many relationships do not naturally translate into a Relational Database Management System (RDBMS) schema. For example, in figure 2.3 a publication can have many authors and an author can have many publications. This is modelled by the relationship between these objects. If these relationships were modelled in a relational database the schema would require additional tables to

manage the relationships as shown in figure 2.4.

Two unique IDs have to be created, `Author.ID` and `Publication.ID` to complete the many-to-many relationship between the `Author` and `Publication` tables. These IDs are connected using an additional table (`PublicationAuthor`) which requires an entry for every relationship between `Author` and `Publication`.

Any program that queries or stores data in the Relational Database Management System (RDBMS) has to join the three tables together before any data can be manipulated. If the tables become large and there are lots of complex relations then there is the potential for a loss of performance. Figure 2.4 shows the additional tables and IDs that are required for the simple object graph shown in figure 2.3.

2.3.3 Data traversals

If the data is accessed by traversing the relations between objects it is often an indication of complex data. For example in figure 2.3 it would be common to locate an author by traversing from an experimental result instance (`T4 Lysozyme`) to a publication (`PCCP`) and finally to an author (`Jack`).

There are two ways to traverse the data shown in figure 2.3 if it is stored using the schema shown in figure 2.4.

Each table has a unique identifier for each row, *e.g.* `Results.ID` which can be used in another table, as a reference *e.g.* `ResultsPublications.ResultsID`. This is called a foreign key. It is possible to combine these two tables producing a row for each referenced identifier, replacing the ID with the data in the row of the parent table. This process is called 'joining' and is used to link tables together so they appear as one large table, enabling the user to reconstruct the relationships between the data.

The easiest way to retrieve the data is to join all the tables together using the appropriate foreign keys, `ID` and then filter the data. This requires that the data access layer understands the additional joining tables and can also convert the request for data traversal into a Structured Query Language (SQL) statement (*Gould et al.*, 1999). Joining all the tables together can be undesirable if they are large and could result in a slow response time.

It is only possible to join all the tables together if all the information is available to create the SQL query. Often data traversals are progressive and the next traverse depends on the previous data traversal. In this case it would not be possible to produce a single SQL query. Alternatively it is possible to query the data for each traversal, return the data and then produce the query to produce the next traversal. Each traverse would translate into at least two queries and a join, one to get the data and relations from the node, `T4 Lysozyme` then another to look up the corresponding relations by joining `ResultPublication` and `Publications`. If there are frequent many-to-many relations this can result in a vast number of database

queries. It also makes the data access layer complex and difficult to optimise.

The query and traversal optimisations of commercial databases can offer better performance than naïve proprietary implementations.

One way to optimise such data retrievals using relational databases is to utilise the features of a triple store, as described in section 2.5

2.3.4 User defined data types

Simple data utilises data types often associated with programming languages, *e.g.* Float, String and Integer but these are often too restricting. For example a publication can have an International Standard Book Number (ISBN) or reference number often consisting of an alpha numeric combination. Using the simple data types provided, it is often difficult to validate the data values. If a system utilises many different custom data types it is up to the user to validate the data during deposition and retrieval.

It is even more complicated if the data types or codes are determined by data type relationships. For example in an object orientated program it is common to use inheritance to provide a data type with a base set of attributes and then override some of these for the specific sub class. In a system that does not support user defined data types it is up to the data access layer to manage these types and ensure that the correct validation code is used.

RDBMS restrict users to a limited number of simple data types. Although it is possible to add checks to ensure that, for example an integer is within a certain range it is more difficult to deal with complex data types. The ISBN example would prove to be more difficult. Any complex data type would have to be managed by the data access layer and it is this detachment of data from the data access code that can cause problems. Especially if the data types utilise inheritance, it can be very difficult to match up a data type with its appropriate code.

There are extensions to permit extensible data types like XML (*W3C*, 2004a), Character Large Object (CLOB) and Binary Large Object (BLOB) but searching these are difficult (*Microsoft*, 2001). CLOBs and BLOBs are often not searchable and there is a limited amount of functionality available. Storing data as Large Object (LOB) breaks the relational structure of a database and bypasses any builtin optimisations (*Leyderman*, 2002).

Many databases such as Microsoft SQL server 2005 (*Microsoft*, 2004), Oracle 9i,10g (*Alapati*, 2003) and IBM DB2 now ship with XML capabilities as standard. These are mainly for converting result sets to XML and for storing XML into relational tables. Additionally for example, Oracle permit the storage of XML in a single column of a table and then enable XML Path Language (XPath) (*Clark and DeRose*, 1999) queries.

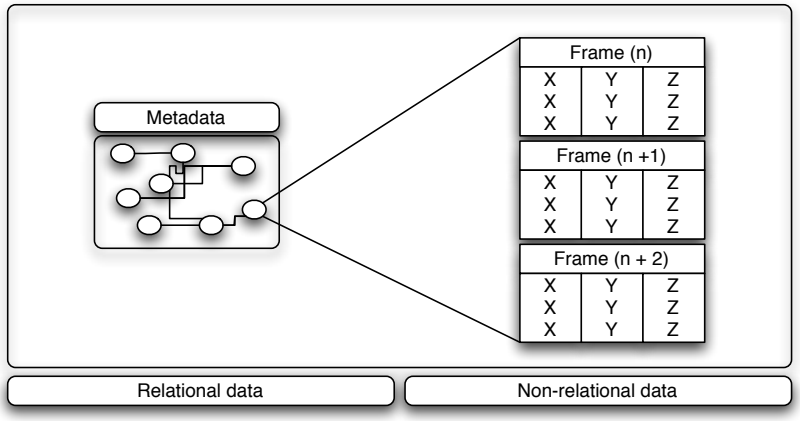

Figure 2.5: Shows how a frame is made up of a series of non-relational coordinates to form frames. The frames link to the highly relational metadata.

2.3.5 Non-relational data

Not all data is complex; it is often the case that the data has very few relationships. For example, figure 2.5 shows the `T4 Lysozyme` experiment result data. It consists of a series of frames each containing `X,Y,Z` coordinates of a collection of atoms. The metadata that describes the experiment is complex and is very relational, but the coordinate data is simple and non-relational.

A frame is a large collection of `X,Y,Z` coordinates, which do not have any relation to other data except that frame in which they belong. This non-relational data is often read serially and does not need any relational data to indicate where the user should obtain the next piece of data. The data should be kept in order to assist with serial reads, otherwise the implied order of frames and atoms has to be maintained using identifiers. If the data is not stored in order then a serial read can involve a large amount of sorting and searching.

If the data is non-relational like the example shown in figure 2.5, it is very difficult to harness the capabilities of a RDBMS. There are two key ways to deposit non-relational data into a Relational Database (RDB) (i) simply add it to a large table using simple datatypes or (ii) add it as BLOBs in a single column. Using simple types creates a large volume of data in a single table which makes joining tables very expensive. Most non-relational data is read in a sequential manner which can cause the database to constantly sort result sets resulting in poor performance (*Sears et al.*, 2006).

BLOBs are non-searchable data types and have no performance increase over storing the data in files. For example, if a non-relational binary file is deposited into a database table as a BLOB, the binary is copied and embedded into the database proprietary data files. Since it is not possible to query and manipulate the data the net result is an embedded version of the original data file which can only be

Table 2.4: Shows the maximum size of a single BLOB, stored in various databases. The maximum number of BLOBs can also be limited by the maximum total database size. There are some exceptions such as PostgreSQL which store large BLOBs as separate files on the local file system.

Database	Maximum BLOB size
Microsoft Access 97/2000	65,535 bytes
Microsoft SQL Server 2000	2 GiB
Microsoft SQL Server 2005	2 GiB
MySQL v4.1	1,048,576 bytes
MySQL v5.0	2 GiB
Oracle 10g R1 Standard edition	4 GiB
PostgreSQL	OS file system limit

retrieved as a whole. This is an unnecessary process as the original binary file is probably more useful as a single file where it can easily be accessed and read by applications.

In cases where the data is non-relational or the data is to be used as a single object, BLOBs or flatfiles should be considered. The maximum size of a BLOB is database dependent and must be considered when flatfiles are not used. The file size is limited by the operating system and is rarely a problem, table 2.4 shows the maximum BLOB size for some key databases.

Using BLOBs can provide a useful mechanism for storing all data in a single location. This is then easy to manage and manipulate using conventional database tools. In some situations this may be appropriate but it can pose problems during a restore. If the database contains many large BLOBs it will substantially increase the restore time in the event of data loss. If the BLOBs are separate the database can be brought on line whilst the bulk of the data is still being recovered. This is advantageous when the data is not *all* required for the system to operate. For example when adding new data, the existing BLOBs are irrelevant.

2.4 Database systems

The term *data base* was first used in the 1960s (*SDC*, 1963) where is was used to describe a collection of entries containing item information; by the 1970s the term had become a single word. The Oxford English Dictionary describes a database as 'a structured collection of data held in computer storage' and according to *Michie* (1968) 'a database is a generalised collection of data not linked to one set of functional questions'.

It is this separation of data from the functions or operations that makes databases useful. The data can be stored and organised without having a complete knowledge of how it will be manipulated in the future. This is suitable for applications where the functions change or are unknown at the time when the data is

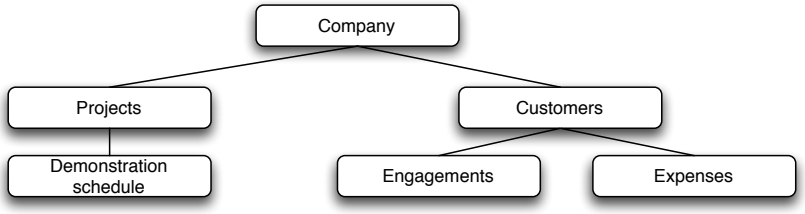

Figure 2.6: Shows an example topology of a hierarchical database.

collected, such as datamining (*Han and Kamber*, 2000).

In the early 1960s Charles Bachman, a great database pioneer, worked on more efficient ways of dealing with direct access storage (*Bachman*, 1965, *Ramakrishnan et al.*, 2002). This work resulted in the network database model and led on to the hierarchical database model. Charles Bachman received the Turing Award in 1973 for 'his outstanding contributions to database technology'.

The first database to gain recognition in industry was the hierarchical database (*CERN*, 2002), which provided a tree-like approach and allowed each record to only have one parent or container and was mainly a mainframe technology. This performed relatively well but the one-record-one-parent approach limited the database's usefulness in modeling real world scenarios.

For example, a project schedules demonstrations which are attended by clients, as shown in figure 2.6. Although both parties are attending the same demonstration it is not possible to link the two tables in a hierarchical database. This results in the data (time, location) being stored twice wasting storage space and making updates difficult.

The relational model was later proposed by *Codd* (1970) but remained of academic interest only until the late 1970s, due to the limitations of hardware. The database market today is dominated by relational databases and section 2.4.2 describes their functionality and implementation.

Since the relational database model was introduced there have only been a few alternatives proposed. The first is the object-orientated database which enables objects with attributes to be retrieved and deposited as a single entity and is further discussed in section 2.4.3. The second is the XML database which attempts to bridge the divide between documents and data and is described in section 2.4.4.

Regardless of the type of database it can be broken into four key areas, each of which is responsible for a particular feature or capability of a database. The four key areas are shown in figure 2.7 and are described in detail below:

- Database and database management systems

  The Database Management System (DBMS) is the program which is used to manage the data in a database (*Ramakrishnan et al.*, 2002). Most often the

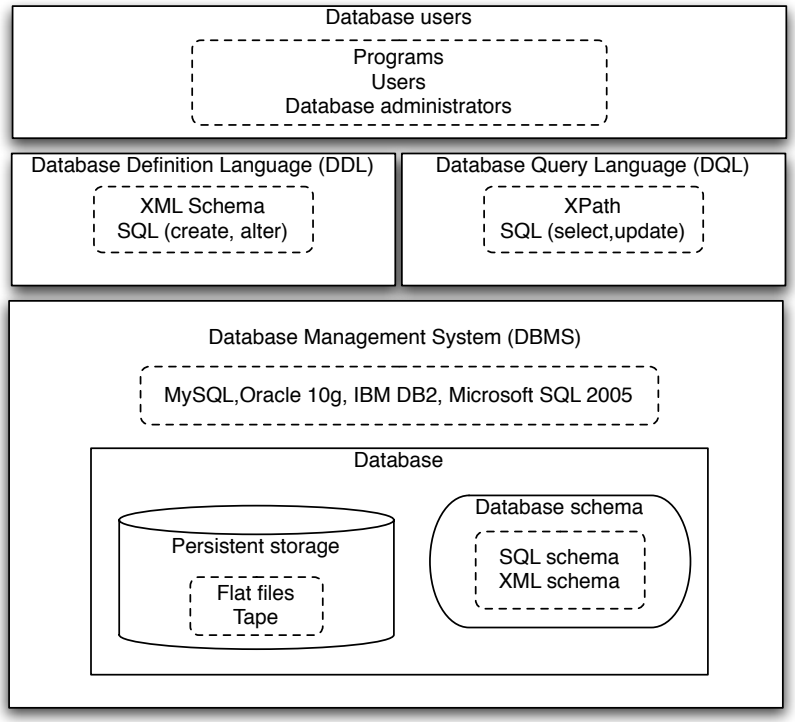

Figure 2.7: Shows the four key areas of database products and examples of each as shown in the dashed boxes.

DBMS is a collection of programs which control the organisation, storage and retrieval of data (objects, records, fields or files) within a database. The database and the DBMS are often referred to as the same thing, but strictly speaking a database is the collection of records and the DBMS is the software which manages the database. Where the meaning is unambiguous this document uses the term database to cover both meanings.

There are some properties that are generally expected from a collection of data before it can be described as a database, however there is no agreed definition. These properties include guarantees about the integrity and quality of the data as well as the ability to share the data amongst a community of users. Most databases have an internal structure or schema, and have the ability to perform some sort of computation on the data using a query language.

- Databases schema

  The database schema is a structural description of the type of data held in a particular database. Designing the schema is one of the first tasks when designing a database. It describes what types of data will be stored, how they relate to each other and provides names which can be used to reference the data (*Adachi*, 2001). In some databases additional items can be added to the schema which do not alter the data but can improve performance. For exam-

ple it is common to add indices to tables or data objects which are frequently used. These indices are a copy of frequently accessed columns, sorted for quick access and linked into any related data.

Example 2.1 shows an XML Schema written in a Data Definition Language (DDL). This schema describes the XML structure of an `Experiment` data type. The schema states what data type is expected, the number of elements in a sequence and the names of the attributes. A sample instance of a compliant XML data document is shown in example 2.3.

- Data definition language

  A DDL is a language for specifying the database schema (*Ramakrishnan et al.*, 2002). These are not standardised and often vary between database vendors but they all provide commands to create and delete databases and tables. Each database then provides a varying degree of commands to manipulate the tables.

  These commands are for creating and manipulating the structure of tables, data types, rows, columns and are not for querying the actual data. Once a database schema is decided the DDL is used to create the internal data structures in a database so that the data can be stored efficiently and optimised for querying by the DBMS.

  An XML Schema is an example of a pure DDL as it only defines the data, unlike SQL where a subset is used for data definition and the rest for data manipulation.

  Example 2.1 shows an XML Schema written in a DDL.

- Query language

  Query languages are computer languages which are used to get data into and out of databases and fall into two categories; Database Query Languages (DQL) and Information Retrieval Query Languages (IRQL).

  DQLs follow a predefined syntax and are like a programming language. In general the user has to have knowledge of the data contained in the database as well as an understanding of the query language syntax. One of the most common DQL is the Structured Query Language (SQL) (*Mattos et al.*, 1999); an example of this can be seen in section 2.4.2. SQL is often used by relational databases but it is important to note that DQLs are not limited to this format and can often look very different. For example section 2.4.4 describes an XPath query for querying an XML file. The overall objective is the same, the query language provides a mechanism for letting users manipulate the data in the database, perform calculations, and process results according to

Example 2.1: Shows an XML Schema which describes the XML structure of an `Experiment` data type. The schema states what data type is expected, the number of elements in a sequence and the names of the attributes. A sample instance of a compliant XML data document is shown in example 2.3.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Experiment">
   <xsd:attribute name="UserID" type="xsd:string"/>
   <xsd:attribute name="description" type="xsd:string"/>
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="Datafile" maxOccurs="unbounded">
               <xsd:complexType>
                  <xsd:attribute name="uri" type="xsd:anyURI" use="
                      required"/>
                  <xsd:attribute name="size" type="xsd:byte" use="
                      required"/>
               </xsd:complexType>
            </xsd:element>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

certain criteria. This process lets users slice data in various different ways whilst leaving the data manipulation up to the DBMS.

In contrast to a DQL an IRQL does not use a formal syntax but rather a best match approach (*Baeza-Yates and Ribeiro-Neto*, 1999). IRQLs apply a weighting or ranking to the results and are often used for searching bibliographic catalogues and museum collection information. IRQLs are also used in search engines where the data is often inconsistent or difficult to manage, for example World Wide Web (WWW) search engines.

Some DDLs are combined with query languages, such as SQL where there are DDL commands to create, alter and delete tables as well as DQL commands to retrieve and insert data.

2.4.1 Common database features and capabilities

The features offered by databases vary between each implementation however there are some standard features. Although these features do not define a database they indicate how advanced databases have become since the 1960s. Any additional database features are used to distinguish database vendors from each other. It is often these features which provide a particular database with most of its appeal.

- Backup

  Database backup is required so that the data can be recovered in the event of a serious failure. More advanced databases have the capability to create a

backup without having to take the database offline. This means that update queries can continue execution throughout the backup process. This is often achieved by logging all transactions that occurred during the backup process and replaying them after a restore. Restoring a database returns it to the exact state it was in when the backup was performed.

Some databases (*Oracle*, 2005a) have the capability to query the database state at a given point in time, without performing a backup. This process is called 'flashback' and it provides a type of online backup as the user can undo any disastrous queries. This is managed by storing the state of the database for any point in time and using the differences between data files or timestamps to retrieve historical data (*Oracle*, 2005b).

In section 2.3.5 we discuss the availability of BLOBs and flatfiles as a possible storage mechanism for large objects of data. This has backup implications as a database backup will include all BLOBs but the flat files remain the users responsibility.

Backing up a database which contains many BLOBs can take time as the volume of data grows. This can become a problem even if incremental backups (*Freeman and Hart*, 2002) are used as a restore can become very time consuming. Many file systems will support full and partial backup utilities. Distributed or clustered file systems and databases can provide quick failover solutions to keep recovery time to a minimum.

- Distribution

  A database can store data in more than one physical database, and still appear as a single instance to the user. Section 5.7.3 shows how distributing data can improve performance, availability and reliability.

- Pre-compiled queries

  When a query is run, an execution plan (*England*, 2001) is generated and then executed. For complex queries this execution plan is time-consuming and can be reduced using pre-compiled queries. A pre-compiled query is often called a stored-procedure (*Henderson*, 2001) and is mainly applicable to common queries as they can be generated in advance. The stored procedure is usually stored within a database which provides opportunities for optimisation.

  The stored procedures are usually written in a proprietary database language such as T-SQL (*Gould et al.*, 1999) and PL/SQL (*Oracle*, 2001, *Feuerstein and Pribyl*, 2005), although some databases also support standard programming languages. Microsoft SQL Server (*Otey*, 2004) described in section 2.7 permits

many commonly used programming languages, Oracle also supports Java (*Gallardo*, 2002).

- Automation

  When the database enters a certain state it is often useful to trigger a job, *e.g.* when a new record is added or when storage or processing capabilities reach a certain threshold. These events are called triggers, and can be applied to many items in the schema (*Adachi*, 2001, *Mullins*, 1999). Triggers provide an efficient event model for databases and have many uses which range from disk space monitors to access permission violations. They can also react to updated data values to trigger business logic processes.

Transactions and concurrency

A database transaction is a unit of work which is completed by a Database Management System (DBMS). Ideally it will execute in an independent, coherent and reliable manner. These transactions often comprise more than one query and it is important that they all are performed without interfering with other concurrent transactions; they are referred to as Logical Units of Work (LUW).

A LUW has a beginning, execution and a commit stage; if either fail, the LUW has to be rolled back. The behaviour of a database rollback is vendor-specific, some rollback the whole query and some just the failed command. Most databases support user transactions and are often called transactional databases.

In order for transactions to successfully complete, most databases attempt to follow a series of rules to ensue these properties. The rules can be summarised by the `ACID` (*Date*, 2000b) properties. These properties are often used when deciding whether or not a DBMS is adequate for handling transactions. Although it is often the case that databases will relax these criteria to varying degrees to improve performance, the overall aim is to prove that LUW are processed reliably.

The Atomicity Consistency Isolation and Durability (ACID) terms are defined below.

- Atomicity

  The results of a transaction's execution are either all committed or all rolled back. All changes take effect, or no changes are apparent.

- Consistency

  The database is transformed from one valid state to another valid state. This defines a transaction as legal only if it obeys user-defined integrity constraints. Illegal transactions are not allowed and, if an integrity constraint cannot be satisfied, then the transaction is rolled back.

- Isolation

  The results of a transaction are invisible to other transactions until the transaction is complete. This prevents one transaction seeing the intermediate state of another transaction, even when operating on the same data. A formal definition states that the transactions should appear serial.

- Durability

  Once committed (completed), the results of a transaction are permanent and survive future system and media failures. It is often the case that all transactions are written into a log which can be played back to recreate a system state before any failures occurred. A transaction can only be deemed committed after it is written and confirmed in the transaction log.

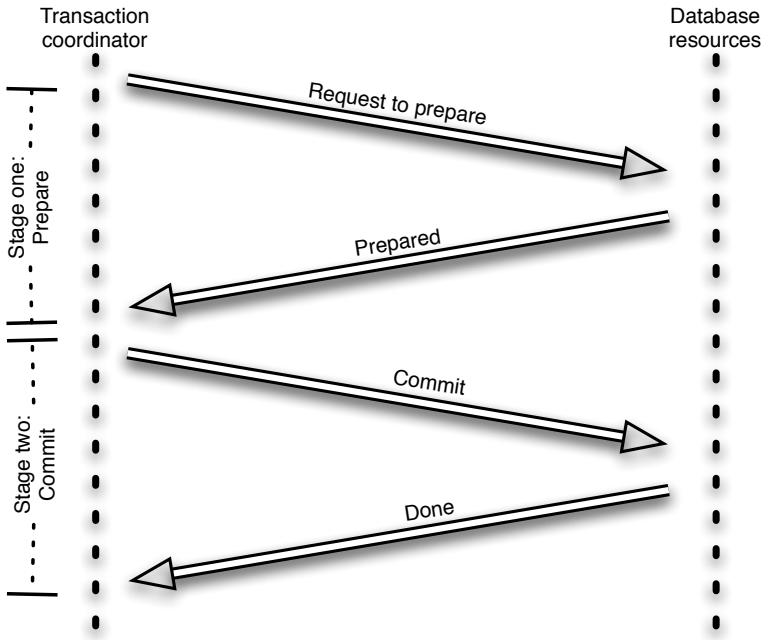

Figure 2.8: Shows the prepare and commit phase of the two-stage commit protocol. The transaction coordinator manages the entire transaction over one or many different databases or resource managers. The first stage ensures that the databases are able to perform the change, `e.g.` acquired write access to tables. The second stage only commits the data when all resources are prepared to commit. This stage makes any changes take affect across all resources. The transaction is finished when all resources confirm they have committed. An error at any other stage will result in the transaction being rolled back (*Gray and Reuter*, 1993).

When data is distributed across more than one database a single commit phase is insufficient to maintain the `ACID` properties. The traditional method for handling distributed transactions is known as the two-phase commit, it breaks transactions into two phases, the prepare and the commit. Figure 2.8 shows the two-phase commit protocol.

Data retrieval

There are two methods of retrieving data from a database, the trivial and the non-trivial.

The trivial method requires that users know the structure of the data and know what information they wish to retrieve. For example, given a record ID the user can retrieve information about that particular record. This is achieved by querying the database tables to access the data required.

The nontrivial case is often called data mining (*Frawley et al.*, 1992). This uses techniques discovered from statistics and pattern recognition to extract data. The aim is to extract information about the data, which was previously unknown but potentially useful (*Hand et al.*, 2001, *Menzies and Hu*, 2003).

This document deals with the creation of databases and repositories with the intention to use them with trivial data retrieval techniques. Although once these repositories are completed it is foreseeable that data mining could be useful.

2.4.2 Relational databases

RDBs (*Codd*, 1970) were developed to ensure that DBMS implementations were independent from any features in the application layer. They use predicate logic and set theory (*Levesque and Lakemeyer*, 2001, *Date*, 2000c) to define a mathematical model of tables and relationships between tables. This permits data to be modelled in a relational fashion. For example, allowing a table to be related to any other table and vice versa.

The RDB stores data in tables and defines their relationships using keys. A key is one or more columns that are in common between two or more tables. The process of defining relations between tables is called normalisation (*Date*, 2000a,d). It provides a set of rules indicating when to split a table and create two tables with a relationship between them.

Figure 2.9 shows two tables, `Experiment` and `Scientist` which have a common key called `ScientistID`. If a user wants a description of all the experiments by a particular scientist, the two tables can be combined using the key column and the data extracted. Example 2.2 shows the SQL required to query these tables.

The queries are written in a proprietary database language, usually SQL (see section 2.4.2) and are interpreted by the DBMS and executed. The results are returned as a table showing the data requested in the query. It is this flexibility to write unforeseen queries against data that provides support for future applications.

One serious limitation of the RDB is its ability to deal with different datatypes. Tables can only store data of a type supported by the RDB and although there are workarounds, storing dates as a string or integer is sometimes unacceptable. Section 2.4.3 shows how object databases evolved to overcome this problem.

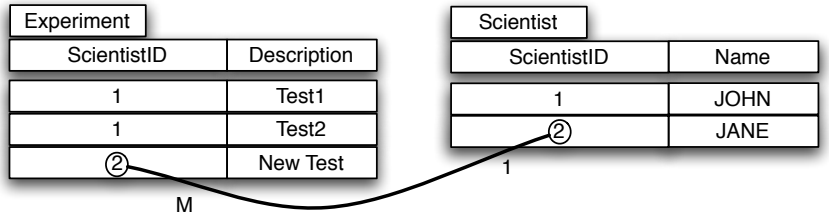

Figure 2.9: Shows two database tables which have a common key; `ScientistID`. The `Experiment` table records information about the experiments carried out. The `Scientist` table records information about the scientists. Storing the data in two tables provides a one-to-many relationship where a single scientists can have many experiments.

Relational databases have been well tuned and optimised over the years and many new features added to increase the ability to deal with more complex data. Today most RDBs support BLOBs (*Microsoft*, 2001) which are suitable for storing pictures and other binary data (*Stephenson*, 2005).

This breaks the RDB data model as BLOBs cannot be queried like native data types, therefore search capability of relations or individual data items inside a BLOB is lost. If BLOBs are used frequently most RDBs tend to suffer from poor performance. The RDB is unable to optimise or search BLOBs thus reducing the effectiveness of a RDB to manage data.

Commonly used RDB implementations are very thoroughly tested and are capable of managing large volumes of data, this often makes them very complicated to operate. Their installation and day-to-day maintenance is often complex enough to warrant the employment of a Database Administrator (DBA). This adds to the cost of ownership and discourages potential RDB uptake.

Structured query language

The Structured Query Language (SQL) originates from work done by *Codd* (1970) and was originally called the Structured English Query Language (SEQUEL) (*Chamberlin and Boyce*, 1974) then later renamed due to a trade mark dispute; the most recent SQL standard is 'SQL:2003' (*Eisenberg et al.*, 2004, *ISO*, 2003).

SQL is both an ANSI and ISO standard but there are many vendor specific variations, the most common are:

- Procedural Language/SQL (PL/SQL) from *Oracle* (2001).
- SQL Procedural Language (SQL PL) from IBM (*Janmohamed et al.*, 2004).
- Transact-SQL from Microsoft (*Otey*, 2004).

Although created for relational databases SQL is used in many other types of databases, for example object databases (see section 2.4.3). SQL aims to provide a

data querying language specifically for managing data although it is not a programming language by definition, and current implementations do not comply with the standards. There are five applications of SQL:

- Data retrieval

  The `SELECT` query is the most commonly used SQL command. It provides a way to return data in a table and restrict the results on certain criteria.

- Data manipulation

  Data manipulation enables users to `INSERT` and `UPDATE` data as well as allowing removal and alteration of existing data.

- Data transaction

  Section 2.4.1 describes how SQL permits the execution of a series of queries and ensures they are all completed or an error is thrown.

- Data definition

  SQL enables users to `CREATE` or `DROP` tables from the schema to define what data should be stored or deleted.

- Data control SQL provides commands to `GRANT` or `REVOKE` user access permissions on schema objects.

Example 2.2: Shows how SQL joins two tables and extracts data from each whilst restricting the result set to contain only data for a single person.

```
1  /* List the items to be returned */
2  select Scientist.Name , Experiment.Description
3     from
4     /* Show which tables to look in*/
5     Scientist , Experiment
6        where
7           /* Join the two tables */
8           Scientist.ScientistID = Experiment.ScientistID
9        and
10          /* Only show the results of a specific person*/
11          Scientist.Name = "Steven";
```

2.4.3 Object databases

Limiting data to simple datatypes has often been seen as a disadvantage of RDBs and has spawned the production of Object Orientated Database Management Systems (ODBMS). An ODBMS integrates database capabilities with Object Oriented (OO) programming concepts (*Versant*, 2001). The result makes database objects appear as programming language objects, accessible from one or more existing programming languages.

The ODBMS extends the object programming language to include transparently persistent data (*Barry*, 2004), concurrency control, data recovery and associative queries, including some of the other database capabilities shown in section 2.4.1. This capability enables database programming from within an OO language such as Java, C++, C# and Python, without the use of a database sub-language such as SQL (*Peterson*, 2001, *Chan et al.*, 1998). They are suited to complex hierarchical data that can easily be traversed and are not well suited to searches across large data sets.

The transparent persistence of objects in the ODBMS provides the ability to directly manipulate data stored using an object programming language. The manipulation and traversal of persistent objects is performed directly by the object programming language in the same manner as in-memory, non-persistent objects. This is achieved through the use of intelligent caching. So unlike RDBs where the user searches for related data, an ODBMS user can retrieve an object which will automatically contain the related data.

For example using a RDB to retrieve data about a user you would first find the user in the `user's` table to obtain the `ID` of that particular user. Having retrieved this information, you can then proceed to get the user's information using the `ID` as an index into other tables for example the `Address` table. This is often the case when you do not know what information will be required next about the user and involves many queries to the database. Using an ODBMS once the user object has been retrieved, all the information about that user can be accessed as properties on that object resulting in the need for only one query to the database.

ODBMS are often used in web sites and XML applications because XML is essentially an object model. It provides support for objects, properties and attributes. The Document Object Model (DOM) is an example of how XML can be used as an object model (*W3C*, 1998).

2.4.4 XML databases

XML (*W3C*, 2004a) can be used to store complex data and most programming languages have the capability of serialising objects to XML files (see section 2.6.4). An XML document can be considered to be a database as it stores data in a structured form where it can be queried using XPath (*W3C*, 1999).

Some database producers (Oracle, SQL Server 2005) have included support for XML in their relational databases and even permit querying of the XML using XPath (*Microsoft*, 2004). Some open source databases like MySQL (*Darrow*, 2004) have additional modules to store and retrieve XML (*DuBois*, 2003). However XML is verbose and slow due to the required parsing and text conversion. Even native XML database performance cannot compete with a RDB (*Borret*, 2004).

Native XML databases either store XML in the original XML file format or re-

Example 2.3: Shows a sample XML data file and the files associated with a single experiment for the user `SJJ`. The `Experiment` has attributes for a `UserID` and a short text description as well as a sequence of `DataFile` types. Each `DataFile` has an attribute for a URI and a data size. An associated XML Schema is shown in example 2.1.

```
1 <?xml version="1.0"?>
2 <Experiment UserID="SJJ"
3 description="Experiment Data Results"
4 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5 xsi:noNamespaceSchemaLocation="file://xmlSchemaExample.tex">
6     <Datafile uri=" file://myhost.example.com/exp1.dat" size="34535"/>
7     <Datafile uri=" file://myhost.example.com/exp2.dat" size="53636"/>
8     <Datafile uri=" file://myhost.example.com/meta.dat" size="4543"/>
9 </Experiment>
```

process it into an internal optimised data structure. In either case the fundamental unit of storage is an XML document which is then searched using either XQuery or XPath.

XML Query and XPath

XPath (*W3C*, 1999) is a language for defining parts of an XML document and is the basis of XML parsing. XPath has over 100 inbuilt functions which often results in XPath being used as a query language.

XML Query (XQuery) is the language designed to query data collections and encompasses the XPath language. XQuery (*W3C*, 2005c) is the SQL for XML enabling users to manipulate data as shown in figure 2.4.

Example 2.4: Shows the XPath for the first `DataFile` entry in the `Experiment` element (line 2). Line 5 shows a simple XPath query to list all data files where the size is greater than 1 GiB. Lines 8 and 9 show an XQuery to list the data files where the size is greater than 1GiB. These queries are based on the XML document shown in example 2.3.

```
1  /* XPath Example */
2  /Experiment/DataFile[0]
3
4  /* XPath query */
5  /Experiment/DataFile[size>1024]
6
7  /* XQuery */
8  for $x in doc("XMLExample.xml")/Experiment/DataFile
9     where $x/size>1024
10    return $x
```

2.5 Triple store

The functionality of databases can be manipulated to operate in ways which they were not designed. For example, a triple store is able to store data in the form of a triplex collection. These take the form *subject-predicate-object* and provide an alternative to a database schema which requires in-depth knowledge of the data

and provides limited extensibility. The triplex is similar to the Resource Description Framework (RDF) standard and many triple store implementations are RDF compliant (*W3C*, 2004b).

A triple store provides a different method for storing data but is often built upon a relational database. The main advantages are that new data predicates can easily be added and the underlying database is easy to optimise for large numbers of entries.

There are many triple store implementations, for example Jena (*McBride*, 2005, *Wilkinson et al.*, 2003) and Redland (*Beckett*, 2001). Scalability is often an issue, although 3Store (*Harris and Gibbins*, 2003) provides a good insight into scalability, demonstrating the ability to manage over 25 million RDF entries.

2.6 File systems

A file system is a database which permits storage of arbitrarily sized binary objects with a limited amount of metadata. They are the quickest method available to archive and retrieve binary objects and are designed to scale without reducing performance. Most operating systems support one or more different file systems and it is for these reasons that flatfiles are often used to store data.

Most file systems limit the metadata (*e.g.* filename, creation date, modified date, owner) that can be stored with a file and this reduces the search capability available to users. It is this limitation that increases the requirement of an additional database to manage the metadata.

There are many different file systems available, varying in performance and features but often operating systems limit the variety of file systems available to the users. Although most file systems store data on disks, others support virtual data and network interfaces, *e.g.* Network File System (NFS) (*Shepler et al.*, 2003). This section looks into the capabilities of some of the more advanced disk based file systems and discusses their capabilities.

2.6.1 Transactional file systems

Transactional file systems log the events for each file in a journal. This journal tracks the transactions of the file system to ensure that updates are committed atomically (*Johnson*, 2001) and is similar to the transaction logs (*Gulutzan*, 2003) that are used in relational databases. They have the ability to execute related changes at the same time to ensure that in the event of failure, the complete transaction can rollback and appear as if none of the changes took place. Reiser and New Technology File System (NTFS) are examples of transactional file systems (*Nagar*, 1997).

Reiser file system

The Reiser file system version 3 (*Gowin*, 2000) is currently the default file system shipped with many Linux (*Newman*, 2003) distributions and the latest version, version 4 has recently been released. Both are journalling file systems which store metadata to avoid file system corruptions. These file systems have all the functionality associated with an advanced file system which protects against data corruptions and responds in a timely fashion.

Version 4 (*Reiser*, 2004, 1984) is capable of supporting a plugin infrastructure which can be used to extend the functionality of files. Currently there are limited examples of how this works but it is similar to the NTFS (*Nagar*, 1997, *Russinovich*, 2000) file system stack which enables an experienced user to add a new layer in the stack, for example a data encryption layer.

Version 4 claims to be the fastest file system available (*Reiser*, 2004) and is capable of dealing with millions of files per directory, even if the files are very small. This is achieved using dancing trees (*Reiser*, 2004) rather than balanced trees (*Stonebraker*, 1993) and ensures that small files are stored more efficiently.

The Reiser file systems have been proven to scale well and operate fast which makes them a suitable choice for managing flatfiles. Additionally the journalling and atomic features are similar to the functionality that most databases provide.

New technology file system

New Technology File System (NTFS) is the standard file system of most Microsoft operating systems since Windows 2000 and originates from Windows NT (*Custer*, 1994, *Nagar*, 1997, *Russinovich*, 2000). NTFS replaces the previous 32 bit File Allocation Table (FAT) file system and is currently at version 5.1. WinFS is expected to replace NTFS when it is released (*Rizzo*, 2004, *Foley*, 2005).

Unlike ReiserFS, NTFS is proprietary which makes interoperability with other operating systems difficult. The file system drive is programmed as a stack of drivers. This enables the transparent insertion of features as they become available. Some of the advanced features supported are described below:

- Quotas

  Version 5 and above permits the allocation of disk quotas for each user.

- Reparse points

  Reparse points are used to replace files, when the file system driver reads a Reparse point it looks up the associated metadata. The metadata is used to point to a filter driver which is then executed. This is commonly used in Single Instance Storage (SIS) where multiple copies of a file are only stored

once in the file system (*William J. Bolosky and et al.*, 2000). Any further copies are replaced with a reparse point.

- Volume shadow copy

  Volume shadow copy creates a copy of modified files by using the copy-on-write driver to ensure that the original data is maintained (*Microsoft*, 2003).

- Compression and Encryption

  The file system has many additional filter drivers which modify the data transparently to the user. For example files can automatically be encrypted or compressed.

2.6.2 Database file systems

The limitations of many file systems lie in the hierarchical management structure. A file resides in a directory and each directory can contain many files. It is not possible for a file to easily reside in many directories. Many file systems permit the linking (*Rosen et al.*, 1999) of files between directories but this has limited uses.

Database file systems are capable of ordering files by their metadata. For example files can be ordered by type, author and date. This enables users to search for files in a similar way to querying a RDBMS. Gnome Virtual File System (Gnome VFS), Be File System (BFS), and Windows Future Storage (WinFS) are all examples of database based file systems (*Nickell and Fergeau*, 2004, *Giampaolo*, 1999, *Rizzo*, 2004).

As well as integrating database capabilities into file systems there are other attempts to incorporate file system capabilities into existing database products. IBM plans to incorporate search capabilities from a research project called WebFountain (*Edwards et al.*, 2002) in a forthcoming version of its DB2 Information Integrator (*Alur et al.*, 2005). Oracle introduced the Internet File System (iFS) into its Oracle 9i database (*Oracle*, 2003, *Alapati*, 2003). It was designed to let people search across databases and data that are typically stored in file systems such as documents and web pages.

Macintosh use the Hierarchical File System (HFS) Plus (*Apple*, 2004) file system and supports the use of metadata searching through a product called Spotlight (*Apple*, 2005b), available for Mac OS X version 10.4 (*Apple*, 2005a). Spotlight automatically builds indices of new and modified files storing a customisable set of metadata about each file. Spotlight does not distinguish between files and other objects such as email and calendar events. This provides users with the capability of searching in real-time for objects such as email, preferences, images and audio.

Windows future storage

WinFS is the codename given the Microsoft's most recent file system expected to ship as part of newer versions of the Windows operating system (*Microsoft*, 2005a) or as a patch for existing versions (*Foley*, 2005). The name WinFS is not expected to be the final product name as it is expected to change when the product is released. WinFS originates from the Object File System (OFS) originally designed for Windows NT 4.0 (*Halfhill*, 1995, *Microsoft*, 2000). OFS was never shipped, and later abandoned.

WinFS uses a database engine to manage data which enables fast searches of files based upon metadata. As the metadata is stored in an application independent database users can search for content, independent of format. Rather than representing a file with a single name, WinFS aims to represent individual data objects, for example emails and address book entries, with searchable context and keyword information. This allows a user to add metadata to any file and then search across that metadata, hence avoiding having to use third party software to produce indices of each data file. This metadata is exposed through two key interfaces, transactional SQL (T-SQL) (*Gould et al.*, 1999) and .NET objects (*Richter*, 2002) which enables users to programmatically search for data files and then manipulate the content.

2.6.3 Distributed file systems

A Distributed File System (DFS) supports the sharing of resources across a network interface providing transparent access to data. A few DFSs examples are the Andrew file system (AFS) (*Howard et al.*, 1988), the Common Internet File System (CIFS) (*Hertel*, 2003) and the NFS (*Shepler et al.*, 2003).

Although some DFSs implement the actual file system, many such as NFS just provide the distributed component. NFS supports many file systems such as Reiser, Ext2, Ext3 (*Tweedie*, 2000) and most other UNIX file systems. Another example is Storage Resource Broker (SRB) which distributes data across different file systems.

Storage resource broker

The SRB is a client-server middleware that provides a uniform interface for connecting to heterogeneous data resources over a network and is developed and maintained by the San Diego Supercomputing Center (SDSC) (*Wan et al.*, 2003, *Moore et al.*, 1996). It enables the storage and replication of data files across many resources and has been shown to scale for large volumes of data (*Rajasekar et al.*, 2002). Each machine runs its own `SRB Server` which is responsible for managing the files in resources.

All the `SRB servers` store data about files in the Metadata Catalog (MCAT).

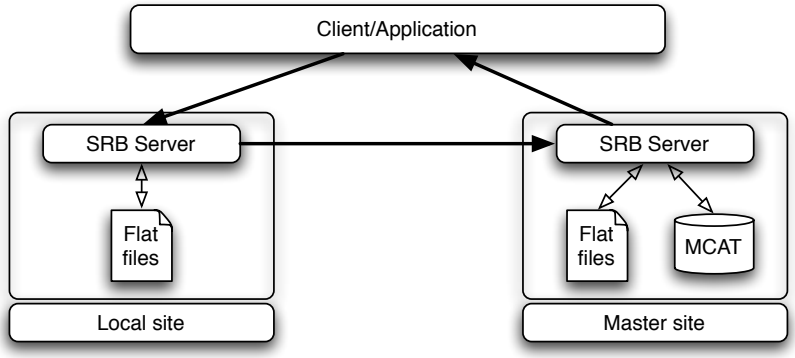

Figure 2.10: Shows how the Storage Resource Broker manages resources. The client requests a file from the local `SRB Server`, the file is located using the MCAT and sent directly to the requesting client.

This is responsible for managing the location, size and replication location of each of the files stored in SRB. Figure 2.10 shows how the SRB infrastructure operates. A client or application always communicates with its local `SRB server`, even to retrieve a remote file. The local `SRB server` first sends the request to the master site running the MCAT database. The location of the requested file is first looked up in the MCAT. The `SRB server` which manages that file is located and returns the file directly to the requesting client. This results in efficient peer-to-peer data retrieval.

SRB supports many other features such as file replication and checksum validation to assist with data integrity and availability. When a file is accessed the most accessible replication is returned. The SRB client provides programming language Application Programming Interfaces (API) *e.g.* Python, Perl , C++ and a Windows and Macintosh Dynamically Linked Library (DLL). There are also Scommands which provide a command line interface using a series of UNIX style commands prefixed with a capital `S` for example `Sls`, `Scd` and `Scat` (*Jagatheesan and Moore*, 2004).

2.6.4 Object serialisation

One of the simplest methods to store data objects is to use object serialisation. This is the process of converting an object in memory into a byte stream (*McMillan*, 2004). It is also known as marshalling (*Lundh*, 2001a), pickling (*Deitel et al.*, 2002) and object persistence. It is available in a wide range of programming languages for example C# (*Richter*, 2002), Python (*PSF*, 2005) and Java (*Chan et al.*, 1998). The main reason to convert an object in memory into a byte stream is to save or transport the object before it is re-instantiated. This can be used to enhance the capabilities of file systems by persisting complex objects in order to manage complex data structures.

Figure 2.11 shows how an object, `Result Set` is converted into a byte stream,

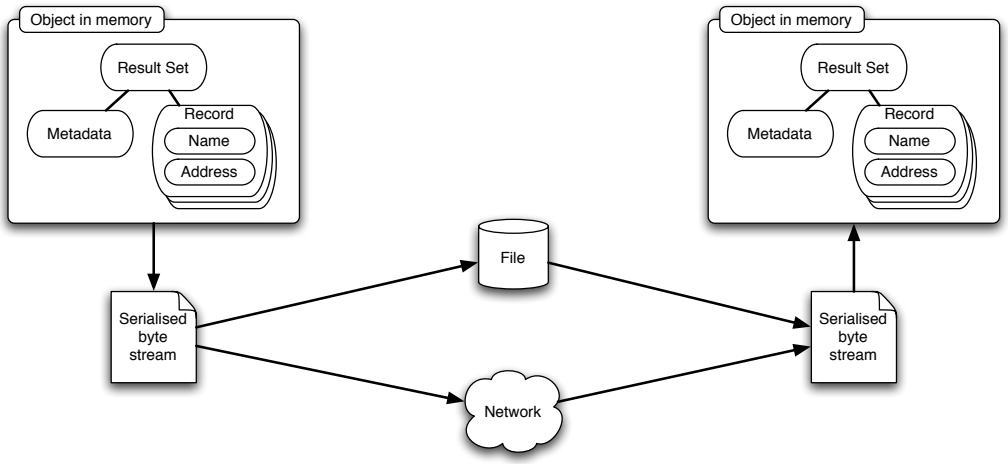

Figure 2.11: Shows how an object in memory is serialised to a byte stream and then re-instantiated later to produce the same object in memory.

saved or transmitted and then re-instantiated. All the child objects of the parent object are also serialised; this results in an almost exact representation of the original object in memory.

Not all data types can be serialised, for example files and sockets, as it is not clear what should happen to them when the object is re-instantiated, *i.e.* there is no guarantee that the same socket will still be active or that the re-instantiation occurs on the same machine.

The internals of the serialisation methods look at the internal state of an object and convert its variables into bytes and repeats the process for any of its child objects. For example in figure 2.11 the `Record`s have name and address fields of type string; these will both end up in the byte stream. When the object is re-instantiated most programming languages require that the class code for that particular object is available. The code must also have the same name as that used when the object was serialised. Serialisation does not take into account the version of the class. This can cause problems if internal variable states have changed as attributes are added and removed as the class code evolves.

Object serialisation is a good method for storing data that can fit into memory and ensures that it can be restored. When there are a large number of objects or an object gets too large to be stored in memory it becomes more difficult to manage. Section 2.4.3 looks into object databases which utilise serialisation to store data with transparent persistence to overcome these problems.

Memoisation

Memoisation (*Michie*, 1968) is a technique which stores the results of computed functions so that they can be used later. This process is used to speed up code execution where functions are frequently used. The advantages become apparent

when the time taken to compute a function is longer than the overhead of storing and retrieving the computed results. When a function is called, a check is performed to see if the function has already been called with the same parameters, if it has, then the stored results are returned. If the function has not been executed then the result is computed and the results stored for later use, before being returned. Some languages like Python (*PSF*, 2005) have modules which support this behaviour (*Martelli et al.*, 2005).

It is important that the function is referentially transparent (*Mitchell and Apt*, 2005) so the stored results remain accurate. A referentially transparent function will always return the same result for a given parameter. Within mathematics most functions are referentially transparent but it is not always clear with computer programs.

For example a function that reads a file and returns a result based on the contents is not referentially transparent if the file contents change. It is more difficult to use memoisation as the behaviour can be changed through global variables or other side effects hence resulting in different results for the same set of parameters.

When a function returns an object it is possible to use serialisation to write the objects to files. This ensures that the memoisation function is not limited by memory requirements. The serialisation of an object into a file is time consuming and therefore only works where the time taken to calculate the results object is greater than the time taken to store and return the serialised results.

2.7 Custom and hybrid databases

Custom databases are produced to address a particular problem or dataset (*Benson et al.*, 2004, *Kelso et al.*, 2004, *Berman et al.*, 2000, *Britton et al.*, 2005, *Szalay et al.*, 2002, *Plante et al.*, 1999). They are optimised to deal with a particular type of data more efficiently than a standard commercial database. For example library and museum databases often require inverted indices (*Litwin*, 1994) to look up related texts based on a search criteria. In this case a custom product with inbuilt indices is more preferable than a standard RDBMS.

Hybrid databases take many forms but are generally a combination of existing database technologies used to leverage the best capabilities of each technology. For example, the object-relational database is a type of hybrid database which provides all the functionality of an ODBMS. It is accomplished with an object to relational mapping layer built upon a relational database. This enables developers to produce applications bases on ODBMS without having to retrain DBAs, as the underlying database is still a RDBMS.

There exist projects which combine conventional database management systems with file systems to produce hybrid database systems. For example the Grid

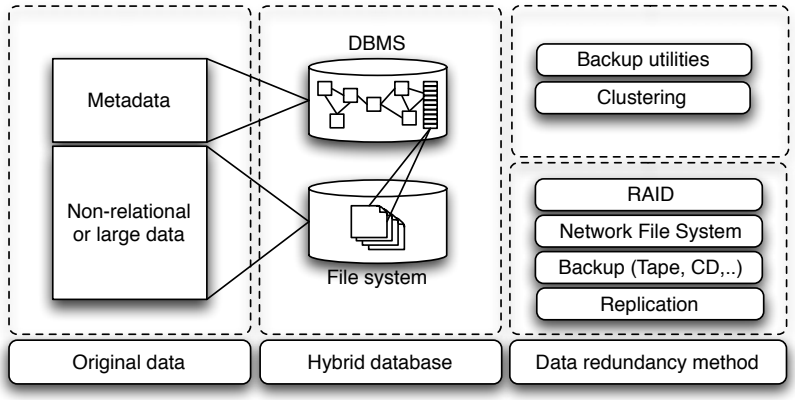

Figure 2.12: Shows an overview of a hybrid database system. The data is split between a database management system and a file system.

Enabled Optimisation and Design Search for Engineering (GEODISE) (*Cox et al.*, 2002) project, which manages its metadata using Oracle and stores data files on a file system. The aim is to leverage on the strong points of each system to produce a database capable of dealing with large volumes of data.

Figure 2.12 provides an overview of a hybrid database and file system approach. The original data to be deposited into the database can be divided into two parts; the metadata and the non-relational or large volume data. The metadata is either present in the original data or has to be produced to describe the original data. This metadata is then deposited into a DBMS such as an ODBMS or a RDBMS. The large volume of data or non-relational data is then stored as a file, or part of a file in a file system. In order for the data to become accessible the file name or location is then stored in the DBMS. This enables users to search the database to locate the location of the data they require then access the file system to retrieve the data.

This approach keeps the DBMS smaller and easier to manage as well as ensuring that the performance is not affected. To ensure data durability on the file system several conventional approaches are used. These vary with a combination of hardware and software backup techniques such as Shadowcopy (*Microsoft*, 2003), RAID (*Vadala*, 2002b), network file systems (*Ghemawat et al.*, 2003, *Butler et al.*, 2004) and data distribution layers.

SQL server and .NET

The .NET (*Microsoft*, 2006) platform enables users to code in over 24 programming languages and the Visual Studio development environment supports advanced debugging features across all supported languages.

Microsoft SQL server 2005 is a RDBMS with many of the standard database capabilities (*Brown*, 2004). It also supports database replication to enable greater scal-

ability and performance, and even permits synchronisation with database instances that connect infrequently. Reliability and performance can also be improved by utilising the fail-over clustering and enhanced multi-instance support.

In addition it has the capability to deal with non-standard data types like XML. Users regularly format metadata using XML, being able to store and search XML is useful. SQL Server has more advanced features for storing, searching and managing XML data.

Microsoft SQL Server has some hybrid database capabilities which deviate from standard database features. For example it is also possible to create database objects using .NET languages such as C# and Visual Basic .NET, thus bringing the database environment into the users preferred language. There are two new objects, user-defined types and aggregates, these help address the issue of user metadata. Users can create their own data types which are then stored in the database. These datatypes can be searched and retrieved using SQL. The user defined aggregates provide a mechanism to compare and sort the user defined data types. For example a datatype `User` can be sorted using a defined aggregate, `greaterthan`. For example the aggregate can return true or false depending on the users `ID` stored within the `User` datatype.

2.8 Grid computing

The Grid is described as a service for sharing computer power and data storage capacity over the Internet (*Foster and Kesselman*, 1999a), Grid computing is the process of utilising the Grid. The intention is to bring together disparate global computers and turn them into a vast computing resource (*Foster et al.*, 2002a,b).

As computational problems quickly outstrip the capabilities of desktop machines it is common to migrate the task to a compute cluster or collection of compute resources. This ensures that the solutions are calculated quicker. As tasks out grow even the biggest cluster, or the biggest available cluster there is a need for more resources, this is where Grid computing plays a role (*Jiao et al.*, 2003). An example of an early Grid application is Search for Extraterrestrial Intelligence (SETI) which utilises machines across the world to process data retrieved from radio telescopes (*SETI@home*, 2006).

The Grid is a type of infrastructure which can take many forms of implementation but there are toolkits such as Globus (*Foster and Kesselman*, 1997, *Foster*, 2005) which provide tools to build and interact with Grids.

An example of a Grid application is Condor (*Litzkow et al.*, 1988), which is a workload management system aimed at managing compute tasks. It can manage resources and schedule tasks on a dedicated cluster or as a cycle stealing role across idle workstations. Condor-G (*Frey et al.*, 2001) utilises many Grid applications and

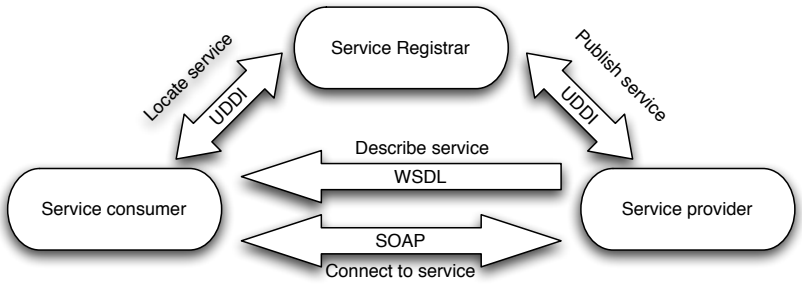

Figure 2.13: Shows the basic web service protocols used to locate and connect to a web service provider.

can operate on resources managed by Globus or as a web service (*Cox et al.*, 2001).

2.8.1 Web services and service-oriented architecture

Web services provide machine-to-machine interoperability over a network, using messages to communicate. The World Wide Web Consortium (W3C) and the Organisation for the Advancement of Structured Information Standards (OASIS) are responsible for the web service standards. They are self describing, platform and language independent and built upon open standards and protocols.

Web services have promoted the progress of Grid computing as they enable the sharing of resources in a discoverable and self describing standard (*Xue et al.*, 2003).

The messages can use many transport and format protocols but it is common for messages to be stored in XML and transported via the HyperText Transfer Protocol (HTTP), mainly as HTTP is permitted through most firewall configurations. The web services are self describing using a machine readable standard such as Web Services Description Language (WSDL) (*W3C*, 2001). Figure 2.13 shows the basic protocols used to locate and connect to a web service provider. The web service provider first registers its service with the service registrar using the Universal Discovery Description Integration (UDDI) protocol. A service consumer then contacts the registrar using the UDDI protocol to locate the service required. Once the consumer identifies a service it acquires a description of the service using WSDL. The consumer then uses the Simple Object Access Protocol (SOAP) to connect to the service provider.

The web service standards are not very mature and have several disadvantages such as the inability to deal with transactions as well as performance issues. There are attempts to support transactions through Quality of Service (QoS) standards (*Menascé*, 2004) but the capabilities fall far short of existing technologies such as the Java Remote Method Invocation (RMI) (*Sun Microsystems*, 2004) and the Common Object Request Broker Architecture (CORBA) defined by the *Object Management Group* (2004).

The performance issues mainly arise from the inefficiency of XML due to the verbose angle brackets and text encoding (*W3C*, 2004a). Standards such as Message Transmission Optimisation Mechanism (MTOM), discussed in section 6.4.2 and other binary formats are attempting to improve the network efficiently of web service messages.

The introduction of web services has greatly enhanced the Service-Oriented Architecture (SOA) programming concept. SOA is intended to scale applications to the enterprise level by ensuring that each service is independent of its implemented technology. The result is a dynamic collection of services, usually distributed across a network (*Jones and Morris*, 2005).

2.9 Discussion

Looking at the available data management technologies it is clear that non-technical users need to leverage existing technologies. We have shown that data volumes are outstripping data processing power, demonstrating a need to utilise data storage and processing capabilities more efficiently.

There are many technologies available to assist with efficient data management, *e.g.* databases, file systems and hybrid approaches. These offer benefits and are suitable for particular types of data but are often ignored by non-technical users.

The complexity of the current data management systems often restricts the uptake. Users have to understand complex systems, and are required to understand the benefits for their particular dataset. Often when users adopt database technologies, their implementation is unsuitable for their particular application. This results in inefficient and often impractical systems.

For example storing large volumes of read once binary data in a database. This makes backups large and slow, often a file system would be more appropriate. Storing ordered data in large database tables can also severely impact performance. Especially if the data has to be sorted every time it is accessed.

There is a need to bring data management applications into the non technical user's environment, where they can utilise the advanced features without needing to learn vast amounts of new technology.

2.10 Summary

It has been shown that storage capacity is growing at a faster rate than processing power. This indicates that there is a need to extend the current data management methodologies so that they can easily manage the information stored within a large dataset.

The current workflow used to manage experimental or simulation data is very

limited and does not promote collaborations. This can result in the loss or recalculation of data. From this we can see that the current workflow requires better organisation in order to efficiently utilise compute resources.

Data storage and retrieval issues are often related to the complexity of data. We outline the reasons which make data complex and address how existing technologies manage these scenarios.

We look at the origins and history of database systems as well as the features offered by file systems. Although databases have matured they are not suitable for all scenarios. This has led to the creation of custom and hybrid databases which aim to address the shortcomings of conventional databases.

There is a need to make the database features more accessible to general scientists in order to improve their current workflow.

Chapter 3

Method

3.1 Introduction

In chapter 2 we provide a background into the current data management technologies available. We describe the data workflow of the existing computational scientists and outline the issues faced when managing large volumes of data.

Based on these challenges, we propose and describe in this chapter a method to assist future scientific work. One of the key issues discussed in section 2.2 is the repetition of work due to the loss or inaccessibility of data. This chapter proposes a method to assist with the collaboration and management of data, particularly scientific data. The proposed method aims to be non-intrusive to the existing workflow but still introduces some of the advantages of databases to scientists. It attempts to transparently bring the world of database features to the inexperienced database user.

We propose a method of associating code with data files by treating them as objects. This provides users with additional functions or operations on a file, without having to provide any code to process the file. Section 3.2 shows how the user's workflow can undergo minor changes to incorporate this method.

In this chapter we discuss the proposed method at an abstract level in an implementation and data domain independent way. Chapters 4, 5 and 7 discuss specific implementations of the proposed method.

3.2 Proposed workflow changes

Figure 2.2, on page 8 shows the existing computational scientists data workflow. Although this workflow has some serious limitations it is unacceptable to change the workflow with which the users are familiar.

In the current workflow the user interacts with the data files using analysis scripts. The user decides which analysis script is suitable for a particular data file. The analysis scripts are unmanaged and often unusable by other users.

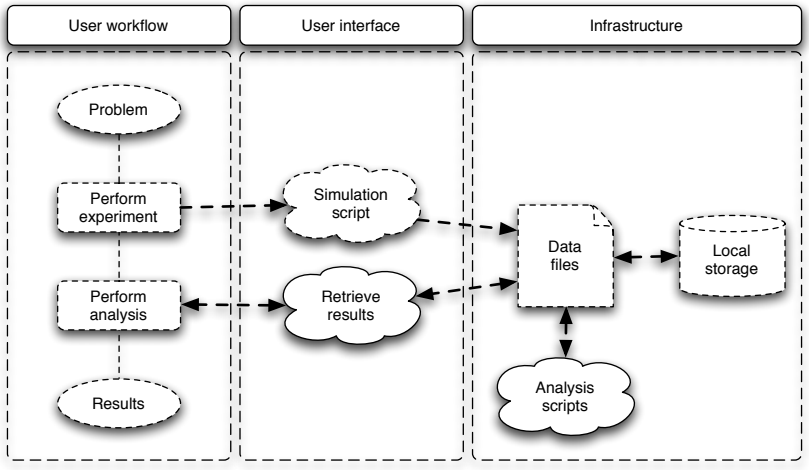

Figure 3.1: Shows the modifications to the original workflow shown in figure 2.2; solid objects represent a change from the original.

Since an analysis script may only be applicable to one data file or data file type the user should not be deciding which script is suitable for a data file, rather the user should be provided with a list of suitable scripts which are appropriate to be run on a specific data file.

We propose to change the workflow's underlying structure to move the analysis scripts into the storage infrastructure, away from the users interface as shown in figure 3.1. When a user wants to analyse a data file they can now query the file to find the associated analysis scripts and then retrieve the results required.

There is only one minor change to the existing workflow. Instead of running an analysis script the user asks the infrastructure to run the analysis script.

The analysis scripts remain unchanged and users can add new analysis scripts into the infrastructure.

3.3 File object method specification

In order to change the workflow as proposed in section 3.2 the infrastructure, currently a file system, has to be able to manage the user's analysis scripts.

Most operating systems are capable of associating an application to a specific data file type, *e.g.* files with the extension `txt` are often associated with a text editor. This enables the user to open the file by 'double-clicking' an icon and relying on the operating system to open the appropriate application which is capable of reading the file.

This mechanism is achieved either by associating the file extension, `txt` with an application or by associating the Multipurpose Internet Mail Extensions (MIME) (*Freed and Borenstein*, 1996) type with an application. Alternatively it is possible to inspect the content of the file to determine its type using a file signature (*Sammes*

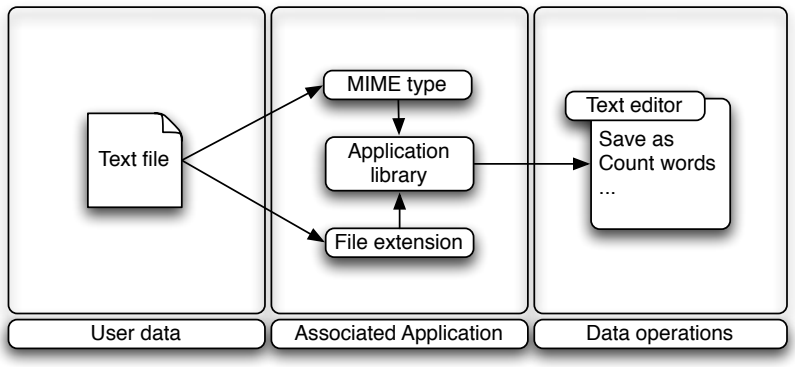

Figure 3.2: Shows how a file extension or MIME type can be used to load an application capable of understanding the data format. In this example a text file is opened using the user's 'favourite' text editor. This has the capability to change the data format and count the words in the file, amongst other operations.

and Jenkinson, 2007).

When a user loads an application associated with a data file, the application can be thought of as a library of functions or methods which are capable of operating on that file. In example shown in figure 3.2 the `txt` file is opened using an editor which has features to count the words or change the data encoding, *e.g.* from ASCII to `UTF-8` (*Yergeau*, 2003).

We propose an infrastructure to extend this common functionality to support custom 'applications', in the form of user defined code. Instead of a data file having an associated application we propose that the data file has associated functions and methods which originate from user code. For example a user with a script capable of returning the word count of a file can associate this code to files of a specific type, *e.g.* any file with the `txt` extension. Any user can then query any `txt` file to see the associated methods, and then execute one of these methods. For example to return the word count of a file by executing a user supplied script.

We propose treating files like programming objects in an infrastructure called the File Object Method (FOM). The FOM associates code with files providing users with the ability to execute these routines as methods on file objects. The aim of the FOM is to extend the usefulness of flatfiles using object orientated programming techniques. This can then be used by hybrid database systems, such as those described in section 2.7 as well as by users who manage data using flatfiles, *e.g.* scientists.

Extending files to appear as objects ensures users execute the correct methods on the correct file types, thus removing the responsibility for users to ensure they have the correct file format. Users can take advantage of methods supplied by other users without having to request access to the code, as the FOM can list and execute the available methods. As the methods are executed using the FOM users do not require extensive Application Programming Interface (API) documentation

and are not required to setup or understand third-party code. This results in a greater opportunity for users to reuse existing code.

There are two ways with which a user can associate methods to files in the FOM. The first is to simply associate code with a single file *i.e.* its path and filename. The second is to associate a `type` with each method or set of methods, and then associate this with file types. This means that users can deposit a file into the FOM, and without any intervention be able to list and execute methods that are associated with that file type, using the data they have just deposited. In the text file example, when a user adds a new text file to the file system they will be able to see that there is a method to count the words in that file.

After consultation with a number of computational scientists, the three most desirable features to make the FOM concept as simple as possible to adopt were identified and are outlined below.

- Format preservation

  Often large volumes of data are compressed using complex compression techniques which make flatfiles an attractive option. This contrasts databases which store all data in the same format *i.e* the native database format. The FOM capitalises on proprietary data formats by leaving the original files untouched. Even if the files are not compressed there is an advantage of storing the files in the user's format. For example, if there was a corruption in the system and the FOM becomes unusable, or the user decided not to use the FOM, the original files are still available in the same directory structure and in the same format as the user's original data. This is similar to the Concurrent Versions System (CVS) approach where the user's data structure remains unchanged. Additionally, users store data in different data formats as they have applications to process the data. Changing the data format can complicate the analysis of data i) as users have to be able to understand the new format, ii) the new format has to provide an interface for users to access the data, iii) defining a data format standard is complex and iv) it is slower to react to application specification changes as many layers require modification. Often existing data formats take advantage of patterns in the information rather than the data, thus making them more efficient at data compression and random data access *e.g.* NAMD (*Phillips et al.*, 2005).

- Code reuse

  The aim is to avoid forcing users to rewrite existing code or create wrappers to get existing code to interface with the FOM. This is achieved by permitting users to add their existing code as an object type in the FOM. This relies on the FOM being able to interpret the methods and its parameters so it can display and execute them appropriately. This issue is discussed for each of

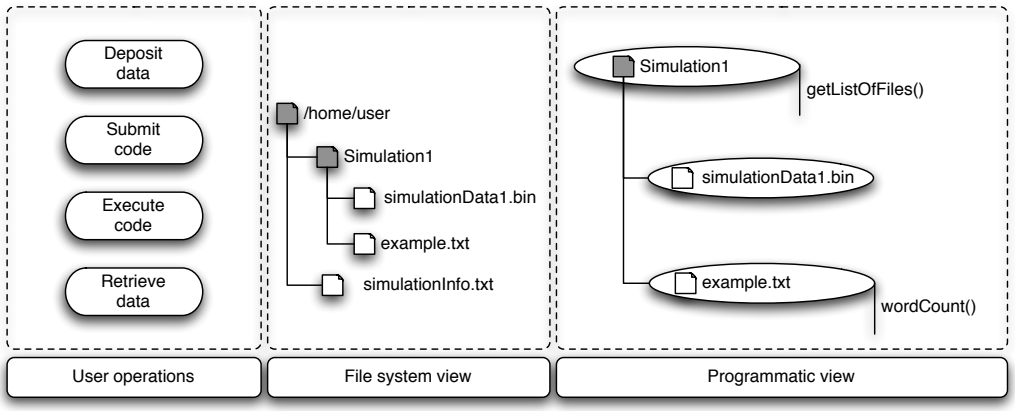

Figure 3.3: Shows the proposed File Object Method (FOM). The user operations provide an interface to the FOM. The file system view shows the users data in its existing format and the programmatic view shows how the data structure is accessible from within a programming environment.

the different implementations.

Reusing code had two advantages i) users can be assured of reliable code and ii) users can utilise code of which they were previously unaware.

- Non-intrusiveness

  To make the FOM as useful as possible it is necessary to bring the database world to the user and not force the user to learn any new skills. The aim is to enable users to continue using their existing code with their existing data files. Although the user's workflow may change, the FOM only extends it with optional steps, leaving the user's existing workflow unchanged.

The three key areas of the FOM are, i) the user operations, ii) the file system view and iii) the programmatic view, of which the file system view remains unchanged.

3.3.1 User operations

Using the FOM, users have four operations available to them as shown in figure 3.3. These operations enable the user to interact with the FOM and are described below.

- Deposit data

  Depositing data is as easy as copying it to a local file system; this should not be any different to the users current workflow. The files get associated with methods based on the file extension, but users have the capability to associate other methods with a file.

Example 3.1: Shows a programmatic Python example of the FOM model. First the user opens the example text file, this creates a data object which supports the FOM methods. The user can list the FOM methods that are available and execute the appropriate method to return the data required.

```
#Load the FOM infrastrusture
import FOM

#Open a text file
 f = open("Example.txt")

#List all the methods available on that text file.
 f.getMethods()
#This output shows that there are three methods available.
 ['wordCount', 'saveAsUTF-8', 'saveAsUTF-16']

#Execute the method to count the words in a text file.
 f.wordCount();
#This output shows that there are 15 words in the text file.
 15
```

- Submit code

  If users want to extend a file object with new methods, they have the ability to submit code that will appear as a method for a specific file type. This operation takes place independently of the data deposition.

- Execute code

  Users have the ability to execute methods on files, this code execution depends on the implementation, but users can initiate the execution from within a programming language or command shell. Either files or data objects are returned to the user.

- Retrieve data

  There are various ways that users can get access to their data. One method is to access it through the file system and not using the FOM. This direct access ensures that any existing user requirements are met. The next is to access it from within a programming language; this is where users will see the FOM methods which are available for each file. This can then be used to retrieve a data object or activate code to produce a data file with the required information.

3.3.2 Programmatic view

Figure 3.3 shows the programmatic view of the example files shown in the file system view. The structure remains the same except the files are treated as software objects. The word count example described in the introduction is displayed as a method on the `Example.txt` file.

A software object is created from the files in the file system and appear within the programming environment as objects with methods. When a user opens a file the FOM methods are added to the file object. Example 3.1 provides an example, showing a Python script opening a file (line 5), listing the methods available on the file (lines 7–10) and finally executing a FOM method on the file (lines 12–15).

There are two types of method: those created by the code submitted by the user and those provided by the FOM framework. For example, the `wordCount` method is from user defined code and the `getMethods` method is provided by the FOM.

The programmatic view is intended as the main interface into the FOM as it enables users to locate data and execute methods to process the data contained within a file.

3.4 Features

There are several advantages of associating methods with file objects and this previously unavailable feature opens up new opportunities for collaboration and optimisation.

The features supported by the FOM are described in sections 3.4.1 – 3.4.8. These features are implemented to various degrees in chapters 4, 5 and 7, and the usefulness of each is discussed in chapter 8. An implementation-independent description of each feature and its requirements are provided below.

3.4.1 User data submission

Users have to be able to manipulate data within the FOM; this includes the ability to add, remove and alter data files. All implementations of the FOM have to permit adding and removing data but there is no requirement for altering data files. As data file alterations can be achieved by removing an existing file and replacing it with an altered file. This is the most fundamental operation of the FOM as it has to permit users to store data using an existing directory structure.

3.4.2 User code submission

The FOM aims to encourage code sharing and collaborations by providing a framework which supports users' existing code. For users to be able to take full advantage of the FOM they have to be able to submit code which can then appear as methods on files or file types.

For this to happen there has to be a mechanism to import users' code. Users are not expected to change their code at all and in the worst case they would have to write a wrapper to ensure that it will work with the FOM.

Ideally there should be an automated mechanism for users to import their existing code, but as this has some security and reliability issues, it is sufficient to have

intervention by an administrator or maintainer of the FOM implementations. The important factor is that users do not write code to fit the FOM but rather they take existing code that would be used on the files and simply add it to the FOM so that it can be accessible to other users.

The programming languages supported by the FOM are based on the implementation. All implementations will support command line programs, which can easily be converted into a method with a lightweight wrapper. All methods that write to the standard output are supported with either returning the data as a datastream (*Powers and Snell*, 2002) or in a file.

Example 3.2: Shows how users can add existing code into the FOM. In this example a user adds a `wordCount` script to all files with the extension `txt`.

```
1 #Python
2 FOM.addScript("wordCount.py", "txt")
3
4 #Shell
5 $ addScript wordCount.py txt
```

Example 3.2 shows two ways for a user to add a word count script into the FOM. Lines 1–2 show how the user can associate a script with all text files from within a programming environment. Lines 4–5 show how this can be accomplished on from within a shell.

Language dependence

If the user's code is language dependent and returns an object rather than writing to a file, then that language has to be supported by the FOM. Support for different languages is a problem as they are often incompatible. We look at several solutions to this problem and the results are discussed in chapter 8.

A FOM implementation support is considered language dependent if it supports the returning of data objects in only one language.

Language independence

It is possible for the FOM to support users' code in more than one language. A FOM implementation is considered to be language independent if it supports many programming languages.

The easiest way to support language independence is to force users to submit code that only supports shell method execution. This is not acceptable as it restricts the applications of the FOM.

It is possible to write a wrapper for user's code to link it into the FOM. It would be unacceptable for the users to write wrappers as the FOM should be transparent.

The FOM has to support the returning of objects regardless of the language in which the user's code is written. Although it is not possible to support any piece of

user code, the FOM aims to support the vast majority of programming languages to manage and process data files.

3.4.3 Code association

Once the user's code exists within the FOM there are various ways which it can get associated with the data. The objective is to associate users' code to data files in such a way that it can easily be executed. Users already have code to process the data files, and the FOM has to create a link between the data and the users' code. There are three different scenarios:

File method association

Users' code can be associated with a single data file on a one-to-one basis. Although this is not the most useful scenario, it does permit users with a mechanism to test the capabilities of the FOM on a single file.

Example 3.3: Shows how users can add existing code into the FOM. In this example a user adds a `wordCount` script to a specific file called `Example.txt`.

```
1 #Python
2 FOM.addScript("wordCount.py", "Example.txt")
3
4 #Shell
5 $ addScript wordCount.py Example.txt
```

Example 3.3 shows how the word count script can be associated with a text file (`Example.txt`) on a one-to-one basis.

If all the files are different and have different code it is a very effective way to ensure that the correct code is executed on the appropriate data file.

File type method association

User's code can be associated with a single data file on a one-to-many basis. Each piece of user's code is given a namespace based on the namespace of the user's classes. This provides the FOM with code types which are then associated with the file types.

Example 3.2 shows how a user can associate a method with all the text files in the FOM.

Users can deposit data and automatically take advantage of the methods associated with that file type. This is where the advantages of the FOM can be seen, a user can benefit from methods provided by other users even on files they have just added, providing that the data type is supported by the FOM.

The FOM has to support the association of more that one user script per file, or file type.

Directory method association

The FOM treats directories and files in the same way, thus making it possible to associate code with a directory. The main use of this feature is to help users manage and order the files within a directory.

For example if all the data files in a directory make up a single data set, the user may want to process all the data in a single directory as a single data set. This can be accomplished by adding a method to the directory object.

Code dissociation

Once data files and code have been associated there may be a need to remove the association. If a user deletes the data file or the associated code then the association will also be removed.

There is also a need to be able to dissociate code and data without having to delete data or code. This can be accomplished by providing the capability to list code associations and remove any unwanted associations.

3.4.4 Dynamic method discovery

Dynamic method discovery enables users to retrieve a list of the methods and their parameters for a given data file. This discovery has to be dynamic to cope with recently added and updated user code. The methods come from two different locations: those associated with the file and those associated with the file type. Example 3.1 shows how the user can retrieve the list of methods for a data file, in this case a textfile. The `wordCount` method is not associated with this particular text file but is associated with all files of this type, these methods are dynamically discovered when the file object is created.

3.4.5 Shell method execution

The shell method execution provides users with the capability of executing methods on files from within an Operating System (OS) shell. This is the simplest implementation of the FOM as it does not require the methods to return programming objects.

The method can either return data or results directly to the shell, appearing on the standard output or by writing them to a file. The FOM does not force any standards on the users' code but it is expected that the files would appear in the working directory, a predefined directory or on the path supplied as a parameters to the method.

The FOM is responsible for executing the method and returning the data to the standard output, subsequently the user is responsible for capturing or storing the data.

Example 3.4: Shows a shell example of the FOM model.

```
1  $ ls -al Example.txt
2    -rwxr-xr-x   1 sjj  sjj  18420 Dec 19 16:19 Example.txt
3
4  $ lsMethods Example.txt
5     wordCount
6     saveAsUTF-8
7     saveAsUTF-16
8
9  $ executeMethod Example.txt wordCount
10    15
```

Example 3.4 shows a command line example. In lines 1–2 the user locates the data file, lines 4–7 list the methods that are available on the selected file. Lines 9–10 execute the word count method and display the result on the standard output.

3.4.6 Programmatic method execution

The programmatic method execution feature is the basis of the FOM as it provides users with the ability to execute methods on the file objects. Executing methods is relatively simple but returning the data to the users is more complex. It is therefore divided into the two following categories:

Results as files

One way to return the results of a method invocation is to write the results to a file. This file is then returned to the user once the method has finished executing.

This method is simple and easy to implement for many languages. In many cases users have code which writes the data into files already so integration into the FOM requires little work.

Results as objects

Often returning the data in files is insufficient. Users require programming objects to be returned which are then manipulated for further analysis. Returning the data as an object is more difficult as it requires the user submitted code to either support the querying programming language or to be able to produce objects which are interchangeable between programming languages.

In example 3.1 lines 12–13 execute a FOM method using Python. This result is returned as an integer object which can then be assigned to a variable and used programmatically.

3.4.7 Cascading methods

The cascading methods feature takes advantage of the directory method mapping to provide users with the ability to bulk execute methods on a set of files. If all the

Example 3.5: Shows how users can use the cascading method feature to map a method to many data files. Lines 5–7 show the files in the `data` directory and lines 10–15 create a directory object and show it has support of mapping. Line 18 executes the mapping method to count the words in each file within the directory and the results are shown in line 20.

```
1  pwd
2 #Show the current working directory
3  /home/FOM/data/
4
5  ls
6 #Show the files in the data folder
7  Example1.tex  Example2.tex  Readme.tex
8
9 #create a directory object
10 d = open(".") # or d = open("/home/FOM/data/")
11
12 #List all the methods available on that directory.
13 d.getMethods()
14 #This output shows that this directory supports a mapping method
15 ['mapMethod']
16
17 #To execute a method on all files in a directory
18 d.mapMethod('wordCount');
19 #This mapping returns the wordcount results for all the files
20 {'Example1.tex': 15, 'Example2.tex': 235, 'Readme.tex': 534}
```

files in a directory have the same method a user may want to execute that method on all files. The cascading feature will enable users to perform this as one single function.

The FOM provides a mapping function which returns a list of method results for every file in the directory which supports the FOM method as shown in example 3.5.

3.4.8 Code quality assurance

By reusing code, users can be assured of the quality of the code and accuracy of its results. The FOM does not offer any inherent means to kitemark code, nor does it set minimum requirements for submitted code. This means that erroneous code can be added to the system.

There are two solutions offered for this problem. One is that users will quickly identify faulty code and be able to remove the methods. This relies on the user community helping to continuously improve the code quality, but does not compensate for malicious code.

The second option only permits the owner of a method to execute it until the method has become mature enough to get released to the community. Establishing when a method is mature and promoting users to publish methods is foreseen as difficult and does not encourage the community spirit of the FOM. For this reason the FOM does not have the ability to restrict methods to particular users or user groups.

In order for implementations to support this feature they must guarantee that

all the code accessible to the user community is free from malicious code.

3.5 Discussion

The ability of the FOM to support many programming languages adds complexity to the project. Wrapping or integrating different programming languages can often be impossible or prone to errors. It is for this reason that this feature will never be fully implemented, to support *any* programming language. To ensure that the FOM is non-intrusive and of benefit to users we aim to support as many of the key programming languages as possible. It is important that users are free to write FOM methods their programming language of preference. It is for this reason that the language independence feature remains in the FOM despite being impossible to fully implement.

The issue of code quality will always be in dispute as it is not possible to check all the code submitted. This feature remains in the FOM specification as the aim is to provide some assurances, i) user's code cannot compromise the host system, ii) user's code cannot corrupt the data stored in the FOM. As most of the security issues rely on the implementation, the aim is to provide as many assurances as possible and highlight the limitations.

The FOM data security comes from the underlying OS, providing a user has permissions to change data the FOM permits the operation.

The FOM does not control the names that users provide for methods and classes, hence naming clashes are expected. This could be overcome by using a namespace similar to that used in Extensible Markup Language (XML) (*W3C*, 1999) or providing an internal FOM name. The end result has to ensure that the FOM is capable of dealing with methods of the same name from different users.

3.6 Summary

This chapter proposes a concept, called FOM, where files are treated as objects, exposing users' code as methods on these file objects. The aim is to enhance the user's data management experience without drastic changes to the user's existing workflow. We discuss how the proposed method can integrate with existing workflow.

The key objectives of the FOM are to preserve the user's data format, and the ability to reuse existing code such that the FOM is non-intrusive to the user.

The FOM provides operations which enable users to deposit and retrieve data into the repository. Once the data is stored in the FOM, users have the ability to submit and execute code on the data.

The FOM objectives and operations are discussed along with an implementation independent description of the FOM. The FOM provides a series of features

which are then described. These features are used in subsequent chapters to discuss different implementations of the FOM.

Chapter 4

File object method prototype

4.1 Introduction

In this chapter we present a prototype implementation, called the File Object Database (FODB) based on the File Object Method (FOM) concept discussed in chapter 3. The File Object Database (FODB) prototype demonstrates some of the key FOM features and provides a discussion into possible future implementations.

We provide insight into a Python implementation and demonstrate the FOM concepts using a simple example. In this example we have chosen to store the user's data in a text file to show clearly how the FOM features operate on the data. The example shows how users can discover methods related to a file, and explains how user code can be submitted and executed using the FODB.

This prototype aims to be a proof of concept for the FOM, used to identify limitations and test the feasibility of the model. From the findings in this chapter we go on to produce production quality improved implementations, as discussed in chapters 5 and 7.

We demonstrate the capabilities of the FODB using a simple example which shows both the FODB implementation and its limitations.

4.2 File object database

The FODB is a proof of concept implementation of the FOM described in chapter 3. The aim is to show how users can discover and execute the methods associated with a file. The FODB aims to test the FOM concept to ensure that it can manage data and users' code efficiently, without disruption to the users.

The FODB has two interfaces: a command line and a programmatic interface. The command line enables users to deposit data as well as user code from a UNIX-style shell. The programmatic interface enables users to programmatically control data in the FODB.

Python (*PSF*, 2005) was selected as the programming language for the FODB

as it is popular within the scientific community. As a scripting language it permits dynamic code execution which is important for executing user code.

The FODB works on a single directory on a local file system. Users allocate a directory for the FODB to store data and are free to add or remove data as they would on any local file system. There is one exception to this, the FODB has to store metadata about files and user code; this is done using a special sub directory in each of the users' directories. This sub directory is always titled `__FODB__` and can be thought of as a reserved directory name.

Those FOM features which are implemented by the FODB are described in sections 4.4.1 – 4.4.7.

4.3 Example scenario

To demonstrate the FODB capabilities let us announce the following example problem: a teacher keeps information about his students in text files, one file per student. Each text file contains information about the student and the marks achieved for previous exams. This file contains the student's date of birth (DOB), firstname (Firstname), surname (Surname) and the marks for three exams (Mathematics, English and Physics).

These files are stored within a directory on the local file system. Example 4.1 shows the contents of the text file `john.student` which stores information about a student called John Smith. This example uses a text file to provide a clear understanding of the internal operations, but will work on binary data in standard or non-standard formats.

Example 4.1: Shows the contents of a single student record (`john.student`).

```
1 SID:    john2004
2 DOB: 28/05/1984
3 Firstname: John
4 Surname: Smith
5 Mathematics:55
6 English:63
7 Physics:B
```

The teacher has a Python script which is used to retrieve information from these text files, *i.e* the mark for a specific exam, or the student's age.

This user script can take two forms, non-object orientated or object orientated. In this example we consider both options for the same example scenario.

Example 4.2 shows a non-object orientated Python script for processing student text files. The example shows three methods, but the script is not limited to these methods.

The first method, `getFullName` (lines 1–3) shows the code to retrieve the student's full name. This comprises of the student's first name followed by the stu-

Example 4.2: Shows the code used to process the students text file (`Student.py`).

```
1  def getFullName(file):
2          ...
3          return Fname , Sname
4
5  def getAge(file):
6          ...
7          return age(DOB)
8
9  def getMark(file,exam):
10         ...
11         return lookup[exam]
```

dent's second name, separated with a single white space.

The second method, `getAge` (lines 5–7) shows the code to calculate the student's current age. This is the age of the student at the time this method is executed and is calculated using the current date and the student's date of birth (see `DOB` in example 4.1)

The third method, `getMark` (lines 9–11) returns the mark for a particular subject. The subject is provided as a string parameter when the method is called. For example calling this method with `Mathematics` as the subject parameter for `John.student` will return the exam mark, `55`.

Example 4.3 shows an object orientated Python script for processing student text files. The example shows a Python class (`Student`) which supports the same methods outlined in example 4.2.

Example 4.3: Shows a user defined class capable of dealing with a data file (`StudentClass.py`).

```
1  class Student:
2          def __init__(self,fileName): #constructor
3                  self.fileName = fileName
4                  self.file = open(fileName)
5                  ...
6          def getFullName():
7                  ...
8                  return Fname , Sname
9
10         def getAge():
11                 ...
12                 return age(DOB)
13
14         def getMark(exam):
15                 ...
16                 return lookup[exam]
```

Let us assume that the teacher has two data files, one for John and one for Fred, stored in a directory titled `StudentData`. In the parent directory the teacher has an information file to store any additional information required; figure 4.1 provides a graphical representation of all the files required for this example.

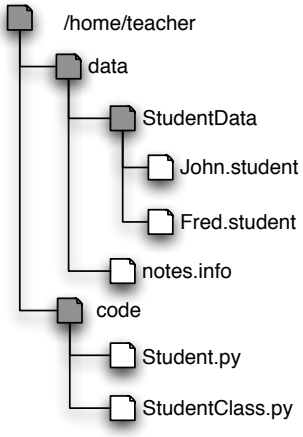

Figure 4.1: Shows the data files required by the proposed example. In the users' data directory (`/home/teacher/data`) we can see a notes file and a directory (`StudentData`) containing the data files for two students John and Fred. The code used in these examples is stored in the teacher's code directory (`home/teacher/code`).

4.4 Features

An overview of the FODB prototype is shown in figure 4.2. Each directory is represented by a `directory` object and each file by a `file` object. The code for each is stored in the `__FODB__` subdirectory. All the user code is stored in the `__FODB__` subdirectory and the data files are stored in the existing directory structure.

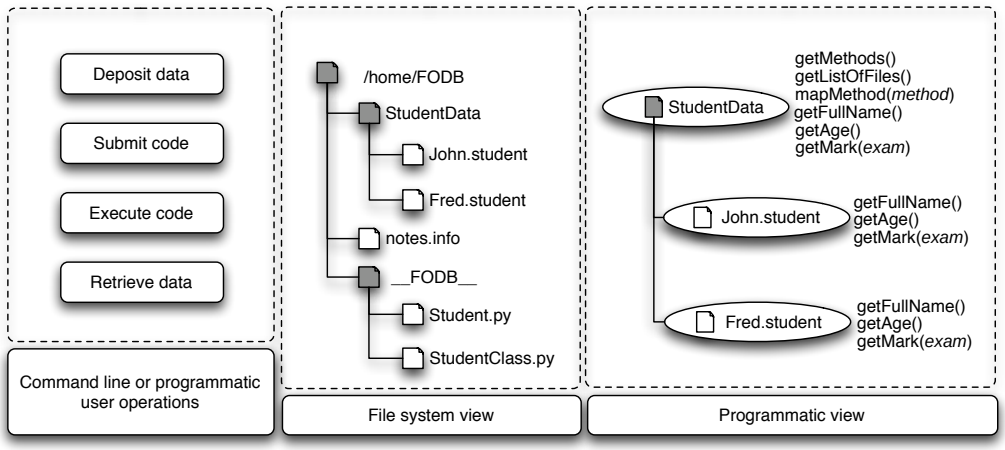

Figure 4.2: Shows the File Object Database(FODB) implementation of the File Object Method (FOM). The user operations provide an interface to the FOM. The file system view shows the user's data and the programmatic view shows how the data structure is accessible from within a programming environment. The file association information stored in __FODB__ directory is not shown for clarity.

When a user opens the directory in Python the `open` method is overridden by the directory object class. This enables the directory to appear as an object with additional methods, within a programming environment.

Example 4.4: Shows how to add and remove data in the FODB. In this example we create the example directory and file structure discussed in the section 4.3. This is accomplished by directly manipulating the file system using standard operating system commands.

```
#Show the users current directory
$pwd
   /home/teacher/data/StudentData

#List the files in the current directory
$ls
   John.student Fred.student

#List the files in the FODB repository (Empty)
$ls /home/FODB
   .  ..

#Create a student data directory in the repository
$mkdir /home/FODB/StudentData

#Copy the data files into the repository
$cp John.student /home/FODB/StudentData/
$cp Fred.student /home/FODB/StudentData/

#Copy a file from a different location
$cp /home/teacher/data/notes.info /home/FODB/

#Remove all the files and directories in the repository
$rm -r /home/FODB/*
```

This also applies to file objects, except instead of returning a file object, the user's class is instantiated and the user's methods are made available as well as the standard file methods. The programmatic view in figure 4.2 shows how a directory appears to a Python user.

When the user at the programming level attempts to open the file, the file object is extended using any code which is associated with that particular data file.

The FOM features described in section 3.4 are implemented in the FODB and discussed below.

4.4.1 User data submission

There are three mechanisms supported by the FODB with which users can submit and remove data from the repository. These provide the ability to i) manipulate the file system ii) manage the FODB from the command line and iii) programmatically manage the data. Using the example provided in section 4.3 we show how the example data can be imported into the FODB repository, see example 4.1. For each example we also show how a user can remove the example data from the repository

Most users are familiar with manipulating the file system. Using the FODB users can add and remove data from the repository using standard Operating System (OS) commands. Example 4.4 shows a user adding and removing data files in the FODB repository.

The FODB provides a command line interface which users can add and remove

data files from the repository. When adding data files, the interface copies the files from their existing location into the FODB repository. To remove data, the files are moved from the FODB repository to a temporary recycling directory. This directory is emptied by the user depending on space requirements. Example 4.5 shows how the FOM command line interface can be used to add and remove data.

Example 4.5: Shows how to add and remove data in the FODB. In this example we create the example directory and file structure discussed in the section 4.3. This is accomplished using a shell command (`FODB`) supplied by the FODB repository.

```
#Show the users current directory
$pwd
   /home/teacher/data/StudentData

#List the files in the current directory.
$ls
   John.student Fred.student

#Import the data using the FODB command.
$FODB import John.student /StudentData/
$FODB import Fred.student /StudentData/
$FODB import /home/teacher/data/notes.info /

#Remove all the data in the FODB.
$FODB remove /StudentData/John.student
$FODB remove /StudentData/Fred.student
$FODB remove /notes.info
```

Users who prefer to use a programming environment to manage data are provided with a Python interface to the FODB. This interface supports the same behaviour as the command line interface and can be scripted using Python; as shown in example 4.6. This is convenient for large imports or automated data management.

Example 4.6: Shows how to add and remove data in the FODB. In this example we create the example directory and file structure discussed in the section 4.3. This is accomplished using a Python module (`FODB`), imported from the FODB repository implementation.

```
#Import the FODB module.
import FODB

#Add the users data files.
FODB.import("John.student" , "/StudentData/")
FODB.import("Fred.student" , "/StudentData/")
FODB.import("/home/teacher/data/notes.info" , "/")

#Remove all the users data files.
FODB.remove("/StudentData/John.student")
FODB.remove("/StudentData/Fred.student")
FODB.remove("/notes.info")
```

4.4.2 User code submission

Users have the ability to import their code into the FODB repository and then associate it with data files. The process of importing code and associating it with data files can be accomplished in one operation. Here we describe the process of only submitting code.

Example 4.7: Shows, both how to add Python scripts into the FODB repository and how to remove existing user scripts from the FODB repository using the command line interface.

```
1  #Show the current working directory
2  $pwd
3   /home/teacher/code
4
5  #Command line user code submission
6  $FODB importCode Student.py
7  $FODB importCode StudentClass.py
8
9  #Command line user code removal
10 $FODB removeCode Student.py
11 $FODB removeCode StudentClass.py
```

Code submission is similar to data file submission, except users cannot provide a location for their code within the FODB. All user code is stored in the __FODB__ directory. Users are not provided with the ability to directly manipulate this directory and have to use either the command line interface or the Python interface.

Example 4.8: Shows, both how to add Python scripts into the FODB repository and how to remove existing user scripts from the FODB repository using the Python interface.

```
1  import FODB
2
3  #Python user code submission
4  FODB.importCode("Student.py")
5  FODB.importCode("StudentClass.py")
6
7  #Python user code removal
8  FODB.removeCode("Student.py")
9  FODB.removeCode("StudentClass.py")
```

Examples 4.7 and 4.8 show how the user interfaces with the FODB repository, to add the user code provided in the example scenario described in section 4.3.

The metadata about users code is small and managed using Python serialised objects. As the volume of data increases the metadata needs to be managed using a database to ensure performance.

4.4.3 Code association

The code association feature enables users to link data files with user code in the FODB repository. All the user's code and associated metadata are stored in the reserved __FODB__ subdirectory. Users can first add either the code or the data files

to the FODB, in this case we assume that both have been added; using the methods shown in sections 4.4.1 and 4.4.2.

The various methods used to associate user code with existing data in the FODB repository are shown below:

File method association

Users are provided with the ability to associate code with data files on a one-to-one basis. Using the example scenario shown in section 4.3, the teacher would want to associate the student Python script (`Student.py`) with John and Fred's data files (`John.student` and `Fred.student`, respectively).

Example 4.9: Shows how the user can associate an existing data file (`John.student`) with a Python script(`Student.py`), providing both already exist in the FODB repository. This example uses the command line interface to the FODB repository.

```
1 #FODB associate <file|extension> <script|class>
2
3 $FODB associate StudentData/John.student Student.py
```

Assuming that both the data and Python script exist in the repository, examples 4.9 and 4.10 show how the user can associate the student Python script with the appropriate data files.

Example 4.10: Shows how the user can associate an existing data file (`Fred.student`) with a Python script(`Student.py`), assuming both already exist in the FODB repository. This example uses the Python interface to the FODB repository.

```
1 import FODB
2
3 FODB.associate("StudentData/Fred.student","Student.py")
```

Once the Python script has been associated with data, users can discover and execute methods on the data using the various techniques discussed in sections 4.4.4 and 4.4.5.

File type method association

In examples 4.9 and 4.10 we show how the teacher can associate the student Python script with the student data files stored in the FODB repository. Associating files on a one-to-one basis is time consuming and does not support the addition of subsequent data files.

Users have the ability to associate code with data files on a one-to-many basis. This is supported through the file type method association feature. It provides users with the ability to associate code with a specific file type. In examples 4.11 and 4.12 we show how to associate a Python script with all student data files.

Example 4.11: Shows the command to associate all student data files in the repository with the student Python script.

```
1 #FODB associate <file|extension> <script|class>
2
3 $FODB associate .student Student.py
```

In these examples if the teacher then deposits additional student data files, they will automatically get associated with the student Python script.

The first parameter of the FODB code association function, shown in examples 4.9 and 4.10 can either be a file name or a file extension. If a file extension is supplied then all files with this extension will be associated with the script provided in the second parameter; as shown in example 4.11 and 4.12.

Example 4.12: Shows the Python code which associates all student data files in the repository with the student Python script.

```
1 import FODB
2
3 FODB.associate(".student","Student.py")
```

Once the Python script has been associated with data, users can discover and execute methods on the data using the various techniques discussed in sections 4.4.4 and 4.4.5.

Directory method association

In the FODB repository, directories and files are treated the same. This means that users can also associate code with directories, on a one-to-one basis. This is particularly convenient for users needing to process many files within a directory.

The directory objects in the FODB are also provided with a set of inbuilt methods. These methods are so frequently used they are included in the FODB. For example all directory objects have a `getFileNames` method to list all the files within the directory. The FODB directory object also supports a `mapMethod` method which takes the name of a method and attempts to execute it on all FODB objects within the directory. Any objects not supporting the method are ignored and the results are returned as a filename result tuple, see example 4.13.

Here we extend the example scenario provided in section 4.3. It is often the case that users have a list of numbers, grades or scores, and they require some statistics about the data, *i.e* mean, median, mode and standard deviation. These are frequently utilised methods and it is not inconceivable that there exists Python code to calculate these statistics on a numerical list.

We can assume that there exists a Python script which takes a numerical array and returns the mean, median, mode and standard deviation of the data. Note that this statistical code is not created by the teacher but by another user of the

Example 4.13: Shows how to utilise the FODB directory mapping method (`mapMethod`) to retrieve the English exam marks for all students in the directory.

```
1  import FODB
2
3  #Create the FODB directory object
4  d = open("data/StudentData")
5
6  #Get an exam mark for all students in the directory
7  results = d.mapMethod("getMark('English')")
8
9  #Show the results
10 print results
11 ...
12 [ ["John Smith" , 63], ["Fred Blogs", 50] ]
```

Example 4.14: Shows the Python class to calculate the statistics of data in a given directory. This class executes the given method on all data files in a directory, these results are then used to calculate the mean, median, mode and standard deviation. The method supplied must return numerical value.

```
1  #Import the FODB and a statistical package
2  import FODB, statistics
3
4  class Statistics(self, directory, method):
5
6          def __init__(self,methodName):
7             self.results = FODB.open(directory).mapMethod(method)
8             self.stats = statistics.calculate(self.results)
9
10         def getMean(self):
11                 return self.mean
12
13         def getMedian(self):
14                 return self.median
15
16         def getMode(self):
17                 return self.mode
18
19         def getDeviation(self):
20                 return self.deviation
```

FODB. Using the `mapMethod` method we can see that the FODB directory object can supply an array of numerical data, see example 4.13.

In example 4.14 we show how a simple Python script can incorporate the Python statistical class and the FODB method mapper. This provides the FODB with a new statistical class which takes the name of any FODB object method, executes it and returns statistical data based on the results. For example users will now have the capability to return the mean, median, mode and standard deviation of all the students' English exam results (see example 4.15).

4.4.4 Shell method execution

The FODB supports the execution of any methods from the command shell. Although all FODB methods are supported there are two limitations: i) the method

Example 4.15: Shows how the teacher can calculate the mean mark for an English exam. This Python script uses the statistical class described in example 4.14 to provide the mean of all the students data which is retrieved using the mapping method described in example 4.13.

```
1  import FODB
2
3  #Open the data directory
4  d = open("data/StudentData")
5
6  #Calculate the statistics of all the English marks
7  stats = d.Statistics("getMark(English)")
8
9  #Show the mean mark
10 print stats.getMean()
11 56.5
```

Example 4.16: Shows the command line execution of the `mapMethod` method. This shows the shell command version of the Python script shown in example 4.13.

```
1  $FODB executeMethod --help
2
3  $FODB executeMethod <file name|directory name> <method> <parameters*>
4
5  $FODB executeMethod data/StudentData mapMethod "getMark('English')"
6
7  [ ["John Smith" , 63], ["Fred Blogs", 50] ]
```

parameters and ii) the method results.

The user's inputs are limited to strings. All the parameters for a method are added to the end of the shell command, in order. Simple objects like lists are supported on the command line. Line 5 of example 4.16 shows how a single parameter is passed to the `mapMethod` method

All the results from FODB methods, executed from the command line are serialised and printed to the standard output. Line 7 of example 4.16 shows how a Python array is returned to the user.

4.4.5 Programmatic method execution

The FODB only supports the execution of Python code and we have provided examples showing how users can invoke methods. In section 4.4.4 we show how data can be displayed using shell commands. The power of the FODB comes from executing methods and retrieving the data into programmable objects. Using Python there the FODB supports two mechanisms to retrieve data, as a file or as an object. Both are discussed below.

Results as objects

Returning method results as Python objects is the most useful FODB feature. Users can execute FODB methods and use the results in subsequent scripts; only Python

Example 4.17: Shows how users can retrieve method results as Python objects. This example is an extension to the script shown in example 4.13. Here we show that the results object can be processed in Python.

```
13 #Count the results
14 print len(results)
15 2
16 #Show the first result
17 print results[0]
18 ['John Smith', 63]
```

objects are supported.

Example 4.17 shows the FODB method returning an array of data. The array is a native Python array of which the users are familiar. In the example we count the data returned and display the first result.

Results as files

Since the FODB only has native support for Python scripts it does not meet the FOM language independence feature described in section 3.4.2 on page 48.

One of the easiest ways to integrate non Python code into the FODB, is to write a wrapper. It is easy to execute shell commands or applications from within Python, however returning complex data is not always possible.

For cases where the returned data is too complex or where the code is not native Python, the FODB supports the returning of data as a file. The file is stored within the FODB making it possible to associate methods with the results files.

Example 4.18: Shows how a user can execute an FODB method and store the results in a file. The file name has to be provided on the command line.

```
1 $FODB executeMethod --file=results.dat StudentData
                                    mapMethod  "getMark('English')"
2
3 $cat results.dat
4
5 [ ["John Smith" , 63], ["Fred Blogs", 50] ]
```

Users have to specify a file name for the results and any data output to the standard output and the standard error are captured and stored in a file, as shown in example 4.18.

4.4.6 Cascading methods

If all the data files in a directory support the same FODB method it is possible that users will consider using the mapping method to execute these methods. It is for this reason that any method which is common to *all* files in a directory will also appear as a method on the directory. This mechanism permits methods to cascade from the directory object, across all data files in the directory.

Example 4.19: Shows how common file methods are also mapped to the directory object. In this example we show the methods available and execute one of the student methods to retrieve the data from all the files in the directory.

```
# Move to data directory
$cd /home/teacher/data/StudentData

$FODB execute listMethods  .
listMethods
getFullName
getAge
getMark
...

$FODB execute getMark 'English'

[ ["John Smith" , 63], ["Fred Blogs", 50] ]
```

For example, a teacher has a directory (`StudentData`) which stores all the student data files. As all the files are the same in this folder, the common methods will appear on the directory object. When the directory method is invoked, it uses the mapper method to execute the required method on all files within the directory. The results are returned using same tuple as the mapper method.

Example 4.19 shows how the methods from the student class also appear on the student data directory.

4.4.7 Code quality assurance

The FODB implementation of the FOM does not impose any restrictions on submitted code. Therefore it is not possible to make any guarantees about the quality of the methods supported within the FODB.

There are some implementation features which protect the integrity of the data. For example because of file system permissions, only the owner can remove or alter the data. All the data in the FODB is readable by all users.

The prototype requires each user to own a user account on the machine running the FODB. Providing the user accounts are properly configured and managed this can help limit the introduction of malicious code.

As users take advantage of the repository each method will be used more frequently. Any errors in existing code are more likely to become visible.

4.5 Discussion

This FODB provides a proof of concept implementation of the FOM. Some of the observations are discussed below:

- The FODB takes the user's code and processes all the methods. When a user invokes a method the user's code is executed, at some point the FODB has

to pass the file name into the user's method. This is accomplished by first looking for a parameter called `fileName`, if this does not exist then the file name is passed in as the first parameter.

This restricts the user's methods or classes but it is unavoidable. In future implementations the user should be able to indicate which parameter should be the file name, either at code submission time or by stating a keyword.

- The FODB limits the user to Python, in future implementations support for further languages would be beneficial. Although an expert user can create Python wrappers for code in many languages.

- The data security model remains the same as all files are stored on the local file system. Owners can control access to files using familiar OS user, group access controls.

- As all the FODB code is stored in the `__FODB__` it can be difficult to link in large applications. When users import code only the one file listed is copied to the FODB repository. This means that all code has to be able to run from either a single file, or has to link with code which the Python interpreter can locate. Since most code will extend over many files the user has to manage the environment path.

- We have demonstrated how the teacher in the example scenario can store and access the existing data without changing the format or directory structure. We then introduced a statistical class capable of calculating various statistical data.

  This demonstrates how the teacher can take advantage of other users' code to process the student's data. The FODB enhances the user's experience by seamlessly integrating third party code. The user does not have to understand how to interface to the statistical class as it appears as a method on the student data files.

- All the information about the code associations and file extensions is stored in the `__FODB__` directory. In chapter 2 we discussed the suitability of storage mechanisms for particular data. Here we are using a file system to store the user's data, to which file systems are well suited. To store the metadata about the data files we should be using a relational database. As metadata is relational and searched often, it is best suited to a relational database.

- The file associations are stored in the `__FODB__` directory.

4.6 Summary

In this chapter we discuss a Python implementation of the FOM, called the FODB. Using an example scenario we demonstrate the FODBs capabilities.

We show how users can submit data and code, either through Python scripts or shell commands. After which we link users' code to either specific files or file types. This enables users to find and execute methods on data files stored within the FODB.

Users can execute methods on data files through shell commands or Python scripts. We demonstrate how the method results can be returned to the user as a Python object or written to a file.

We demonstrate how the FODB can be used to enhance the user's experience. In this example we show how the user can take advantage of another user's code to obtain previously unavailable data results.

The FODB provides a proof of concept from which we discuss the advantages and disadvantages of the FOM model.

Chapter 5

BioSimGrid

Preface

The BioSimGrid project is a collaborative project involving the work of others of which I developed and implemented the flatfile storage layer, consisting of controlling software and data structures. This includes the integration of BioSimGrid with the Storage Resource Broker (SRB) and the development of the deposition module for the simulation package.

The work involving the analysis tools and metadata schema are the work of others. Some parts of this work are published in the papers cited in the *Authors declaration*. The SRB integration software was presented at Europython 2005 (*Johnston*, 2005b).

5.1 Introduction

As Molecular Dynamics (MD) simulations are becoming more and more popular there arises a need to manage the simulation results, especially as they are time consuming and require expensive compute cycles. BioSimGrid (*Ng et al.*, 2006, *Murdock et al.*, 2002, *Tai et al.*, 2004) is establishing a worldwide repository for simulation results using Grid (*Foster et al.*, 2001, *Foster and Kesselman*, 1999b) technologies to distribute and manage the large volumes of data. It aims to produce a transparent repository to store distributed simulation results as well as providing a framework to perform analysis (*Tai et al.*, 2006). Each analysis tool is designed to encourage user contributions and is based on an extensible framework. BioSimGrid is already enabling new science by providing a mechanism for scientists to compare simulation data which originates from different laboratories in different formats (*Woods et al.*, 2005).

This project provides a suitable platform to experiment with some of the concepts described in chapter 3, in a production environment. This chapter describes BioSimGrid from both a user's viewpoint as well as discussing the underlying in-

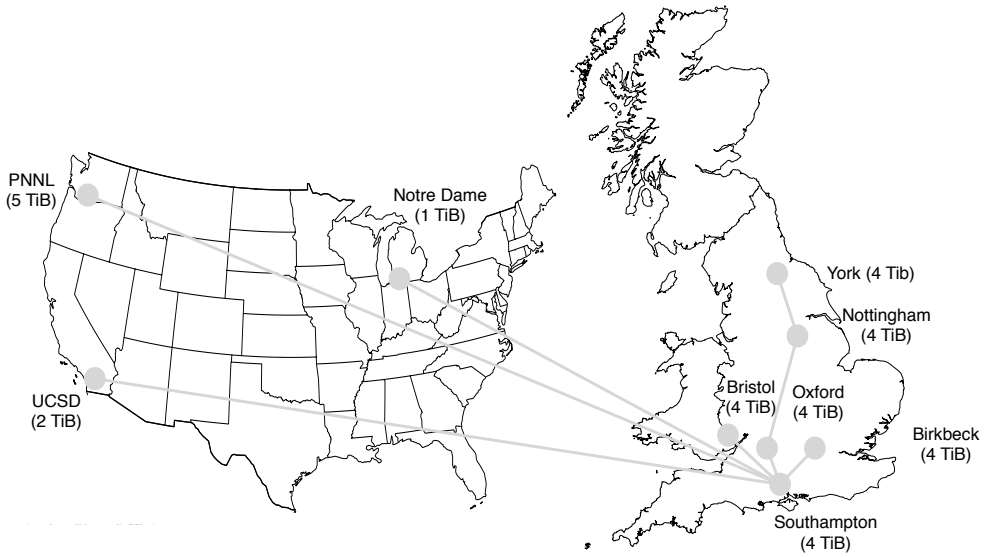

Figure 5.1: Shows the location of the six collaborating institutions in the United Kingdom and the three collaborating institutions in the United States of America.

frastructure. It then goes on to describe how features from chapter 3 have been integrated into BioSimGrid and discusses the advantages and disadvantages.

5.2 Motivation

X-ray crystallography (*Drenth*, 2002) and Nuclear Magnetic Resonance (NMR) spectroscopy (*Silverstein et al.*, 2005) are two common techniques used to gain an insight into the molecular structure of many molecules and materials.

X-ray crystallography uses the diffraction of X-Rays passed through a sample to determine the molecular structure. This process requires the growing of a crystal for analysis, which is then frozen in liquid nitrogen to reduce radiation damage and thermal motion. This sample is then subjected to X-rays which diffract off the electrons in the crystal. A crystal is required to ensure that there is a regular structure to diffract the X-rays as a single electron is insufficient. The diffraction pattern emitted is recorded and used to built up a diffraction map of the electron density. This is then used to calculate the molecular structure (*Allen et al.*, 2004).

NMR spectroscopy uses the magnetic properties of the nuclei to build up a model of the structure. In a constant magnetic field the nuclei resonate at a frequency dependent on the strength of the magnetic field, this frequency is converted into a field independent value called the *chemical shift*. Each nucleus has a different chemical shift, providing a means to identify different atom types. NMR is the most commonly used method to determine structure in modern chemistry and is capable of dealing with complex molecules.

In the biological world these techniques are used to determine the structure of many molecules which are then stored and distributed throughout the community. Often once molecular structures are determined, they are stored for use by other scientists, *e.g.* in the Protein Data Bank (PDB) repository (*Berman et al.*, 2000).

Although these techniques have been used for over 50 years to determine the structure of many biological components they have some drawbacks. They only provide a snapshot of the molecule at a point in time and do not provide a means to analyse the motion of the molecule. This is partly due to the temperature at which the samples are subjected as most biological molecules are active around room temperature.

This has given rise to biomolecular simulation methods such as MD which are capable of simulating time-dependent molecular models (*Haile*, 1997, *Stone et al.*, 2001). MD uses classical equations of motion to predict the motion of the atoms within a molecule.

As the position of each atom is calculated at every timestep using its previous location and velocity, MD calculations are often limited by compute resource capability. If the timestep is too small the calculation will quickly consume too much processing power, however if it is too large it becomes too inaccurate.

MD simulations require large volumes of data storage and compute resources. Once calculated, they become valuable data that can be analysed by many different techniques. Currently as MD simulation data is generated within research laboratories throughout the world, they are stored and analysed locally. Once any findings are published the data is then archived. In reality this often results in the loss of the data. As the research laboratories have no means to share the simulation data many MD calculations are repeated, consuming one of the most valuable resources, computing power. Section 2.2.2 discusses the merits between re-computing and storage and shows that storage is often not the optimal solution, however in the MD situation the compute resources are the limitation and therefore storage is clearly sensible.

For the biochemical community to discover tomorrow's solutions using today's hardware, better compute resource management is required. This begins with the community sharing existing MD simulation results, so that other laboratories can perform analysis without first having to perform a simulation. This collaborative approach also provides a kitemark process by which publication findings can be validated using the same simulation data but with different analysis procedures.

This lack of collaboration ability means that the discovery of simulation data in different institutions is very difficult. In addition it is nearly impossible to compare sets of simulation data across institutions, as one set of data would have to be copied to the other site. Even once the data resides at the same location it is common for different institutions to use different data formats, thus making a direct comparison very difficult.

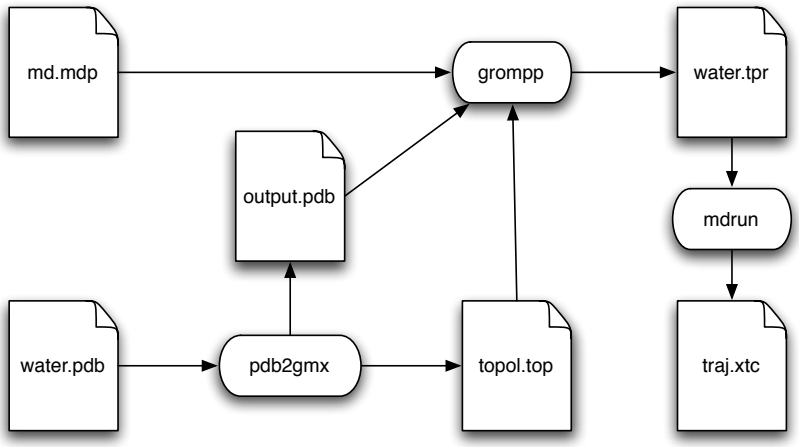

Figure 5.2: Workflow to simulate a sample molecule to produce a trajectory suitable for analysis.

BioSimGrid addresses these issues by creating a freely available globally distributed repository to store simulation results. BioSimGrid allows users to deposit simulation results, perform analysis and provides a kitemark procedure to guarantee the quality of the data. This provides users with the ability to apply new science to the existing data sets.

5.3 Example simulation

In this section we show how a chemist performs an MD simulation using a simple example. We simulate the behaviour of water molecules over a period of time and show which files are used and produced throughout the simulation. This simulation is performed for five timesteps and for each timestep the positions and velocities of each atom are recorded. The simulation starts with a structure of water molecules which are then simulated to produce a trajectory. The trajectory can be thought of as a video showing how the molecules move over a time period. The simulation process is shown in figure 5.2 and described below.

- Step 1

  The user starts with two files, the `md.mdp` and the `water.pdb`. The first contains the parameters for the molecular dynamics simulation application and is created by the chemist. For example it determines the number of steps to simulate and the time, temperature and pressure for each step.

  The second provides a 3D structure of the molecule (typically a peptide, protein or nucleic acid) to be simulated. These are obtained using experimentation such as X-ray crystallography (see section 5.2) and are available online from sites such as the PDB website (*Berman et al.*, 2000).

The scientist first generates a topology file from the coordinates contained in the `water.pdb` file by calling the `pdb2gmx` executable, as shown in example 5.1. This provides two output files the `output.pdb` and the `topol.top`. The first contains the *corrected* set of coordinates based on the `water.pdb` file, *i.e.* any missing hydrogen atoms have been added. The second contains the force field parameters and type of molecules present. These files describe the topology of the molecule to be simulated.

Example 5.1: Shows how a user generates a topology file from a PDB file.

```
#Input water.pdb, output topol.top

$pdb2gmx -f water.pdb  -o output.pdb
```

- Step 2

  The `md.mdp` , `output.pdb` and `topol.top` files are then combined into a single simulation file, `water.tpr`. This file contains all the information required to perform the simulation and is generated by calling the GROMACS pre-processor executable `grompp`, as shown in example 5.2

Example 5.2: Shows how a user combines the input files to produce a single file containing all the parameters to perform the simulation.

```
#Generate .tpr file

$grompp -p topol.top -c output.pdb -f md.mdp -o water.tpr
```

- Step 3

  The `water.tpr` provides the starting simulation state and parameters to initialise and run the simulation. The simulation is performed using the command shown in example 5.3. The simulation state is written to the `traj.xtc` file every timestep. The `traj.xtc` provides the coordinates of all the atoms over a period of time.

Example 5.3: Shows how the user performs the simulation.

```
#Peform simulation

$mdrun -s water.tpr
```

- Step 4

  Once users have the simulation or trajectory files they can then be analysed using applications such as VMD , NAMD and GROMACS (*Humphrey et al.*, 1996, *Phillips et al.*, 2005, *Berendsen et al.*, 1995). It is often the case that the

trajectory is analysed many times and it is the goal of BioSimGrid to produce a repository to store these trajectory files.

For example, using the `traj.xtc` file produced from the simulation and the input structure file (`output.pdb`) a user can visualise the frames stored in the trajectory. Example 5.4 shows how to visualise the trajectory and figure 5.3 shows the first frame in the simulation.

Example 5.4: Viewing the data output from the simulation.

```
1 #VMD structure file and trajectory coordinates
2
3 $vmd output.pdb traj.xtc
```

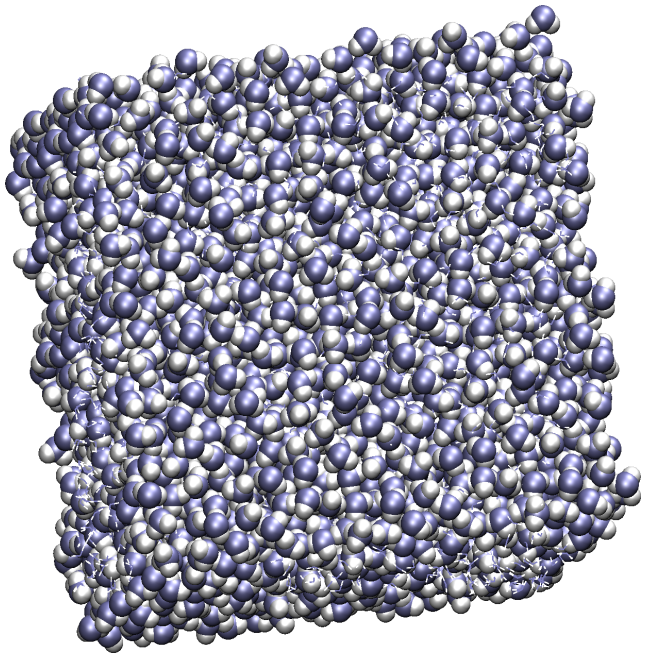

Figure 5.3: Shows the first frame of the example trajectory. The frame is comprised of water molecules.

5.4 BioSimGrid overview

The BioSimGrid project (*Johnston et al.*, 2006b, *Boardman et al.*, 2006) attempts to address the issues associated with managing simulation data and collaborating with biological and chemical communities across the world (*Wu et al.*, 2005, *Tai et al.*, 2004). The aim is to produce a worldwide repository to store and analyse simulation results. More importantly it provides public access to the analysis data described in publications.

BioSimGrid is a collaboration between six universities in the United Kingdom (Birkbeck College – University of London, Birmingham University, Nottingham University, Oxford University, University of Southampton and York University)

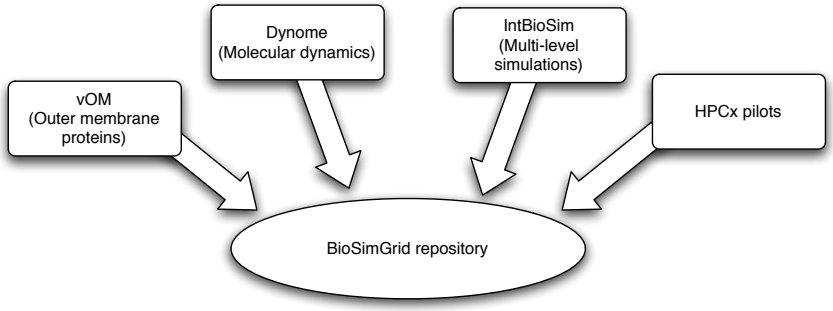

Figure 5.4: Shows some of the projects that relate to BioSimGrid. Virtual outer membrane (vOM) generates data of outer membrane proteins (*Bond and Sansom*, 2004). Dynome aims to simulate the dynamics of a broad range of the most important protein shapes or folds (*Essex*, 2005). IntBioSim aims to develop a framework with which to conduct biomolecular simulations, using BioSimGrid as the underlying storage repository for the simulation data. There are other HPC pilots which can potentially utilise BioSimGrid (*Bush and Sunderland*, 2005, *Sansom*, 2005).

and three institutions in the United States of America (University of California, San Diego (UCSD), Notre Dame and Pacific Northwest National Laboratory (PNNL)). Each site has its own storage and processing resources to provide quick access to simulation data providing over 26Tebibyte (TiB) of storage (see appendix A). The repository is currently a working prototype open to users across the collaborating sites and used for biochemistry research (*Woods et al.*, 2005). Figure 5.4 shows how BioSimGrid relates to some of the ongoing projects such as Dynome (*Essex*, 2005), IntBioSim (*Sansom*, 2004), vOM (*Bond and Sansom*, 2004) and high performance computing pilots High Performance Computing (HPC) (*Sansom*, 2005, *Bush and Sunderland*, 2005).

We provide a brief description of the key features offered by BioSimGrid as shown below:

- Distributed storage service

  This provides BioSimGrid with the ability to manage and utilise storage resources across the world. As demand increases storage can be added at any location and is not dependent on a single storage center. This assists with data management optimisation, as the data can reside close to where it is required.

- Transparent data access

  Users are oblivious to the location of the data and an analysis script can run on any BioSimGrid site. This abstracts the user from the underlying data structure and offers accessibility to simulation results located throughout the BioSimGrid repository.

- Multiple data format deposition

  There are many MD software packages, as shown in section 5.5.2, each with their own data format and capabilities. The BioSimGrid deposition manager

provides an extensible interface to support plugins for each format, currently all the key MD software packages are supported. This provides support for future and private data formats.

- Analysis tools

  The BioSimGrid project provides a set of analysis tools, based on an extensible interface. The base analysis tools double up as examples for users who want to write their own tools. The aim is to provide a platform to leverage community contributions by providing a simple mechanism to write and share analysis tools.

- Cross-site comparison

  The repository facilitates sharing of simulation data through a uniform interface which enables users to compare data regardless of its original format and location. This ability to discover and compare simulation data from different institutions provides the community with new, previously unobtainable capabilities.

- Data quality assurance

  BioSimGrid provides a kitemark (*Murdock et al.*, 2006) capability for simulation data which provides users with assurances about the quality of the simulation data.

- Data availability and persistence

  There are assurances about the availability and persistence of deposited data, ensuring that data is not lost and is available to users when required. This is accomplished using data redundancy on a distributed data grid running on fault tolerant hardware (see appendix A.1).

5.5 User perspective

Users are provided with tools to store and analyse trajectories in the BioSimGrid repository. This is provided using the scripting environment described in section 5.5.2. In addition there is a web portal which provides a demonstration of the capabilities of BioSimGrid (*Wu et al.*, 2004a,b). It enables users to search existing simulation data and execute simple analysis scripts, displaying the results in a web browser (see figures B.1—B.5). The web portal is an extension to the underlying scripting environment and is intended as an introduction to the scripting environment.

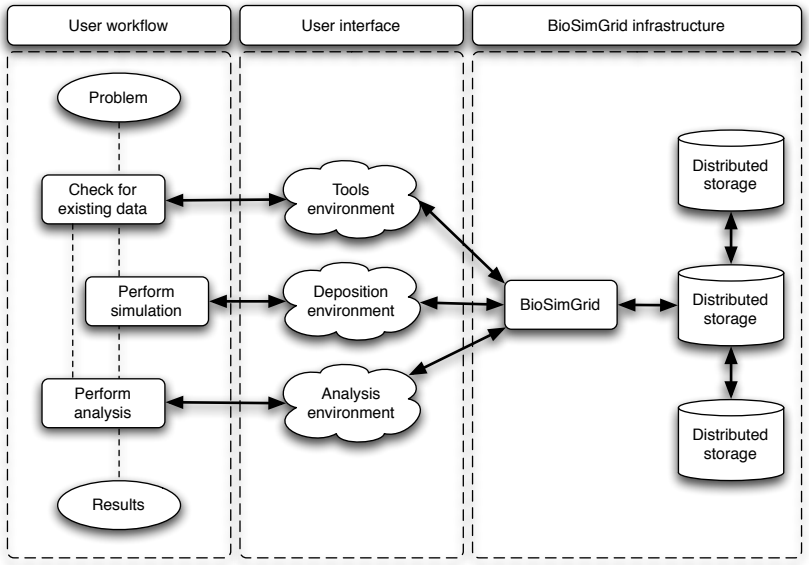

Figure 5.5: Shows the user's workflow and how each item relates to BioSimGrid components.

5.5.1 Work flow

The current work flow results in the inaccessibility and repetition of simulation data and is the motivation behind BioSimGrid as described in section 5.2.

Before a scientist performs a simulation he or she can first check to see if it or something similar already exists within the repository. If the data already exists then it can be analysed, otherwise the simulation will need to be performed. After performing the simulation a user can then deposit the trajectory into the repository regardless of location or storage requirements. Depositing simulation data means that it can easily be shared amongst other users. A user can then perform analysis on the data either by using the builtin analysis tools or a custom tool.

Most users have their own analysis scripts, which can be added to the BioSimGrid analysis environment, as show in figure 5.5.

5.5.2 Scripting environment

BioSimGrid is programmed in Python (*PSF*, 2005) as it is widely used by the collaborating sites and commonly used within the biochemical community. BioSimGrid provides a Python scripting environment by adding modules to the existing Python environment resulting in a powerful, familiar Problem Solving Environment (PSE).

The Python PSE provides users with the many of the tools, packages and modules to solve problems. This is important as it provides a single environment capable of solving complex problems (*Black et al.*, 2003). Python also supports many third-party applications and can interface to other programming languages such as C.

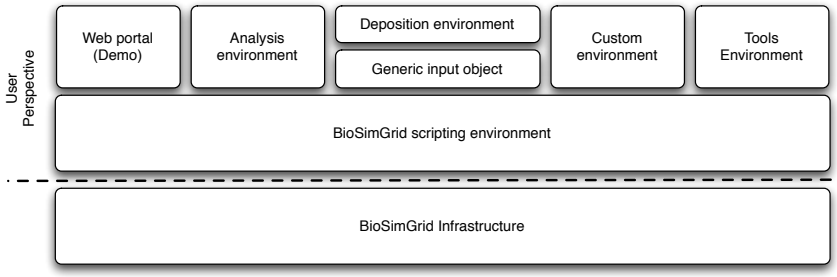

Figure 5.6: Shows how the user's perspective is separated from the BioSimGrid infrastructure. All the user components are built upon the Python scripting environment. This ensures that the extensibility of BioSimGrid is separated from the underlying infrastructure. The existing components are themselves examples for users wishing to customise or create new components.

The BioSimGrid scripting environment is the only interface exposed to the users, it supports all the capabilities required to access and manipulate the data; all the other user components are built upon this. Community support and contributions are supported through the clearly defined interface.

Example 5.5: Shows the BioSimGrid scripting environment. After importing the necessary modules and setting up the user's credentials, the user is free to retrieve any coordinate or metadata they require using the supplied methods. This example shows how to retrieve the coordinates of the atoms in the first frame of the sample trajectory. It also shows how easily the metadata about the frame can be retrieved, showing how to retrieve the temperature of the first frame.

```
1  from BioSim.DataRetrieval import FrameCollection, FCSettings, Frame
2  from BioSim.Settings import UserSettings
3
4  #Setup the user credentials
5  USettings = UserSettings("Bob")
6
7  #This selects frames 1,2,3 from trajectory BioSimGrid_GB-STH_1
8  datasettings = FCSettings(USettings,[["BioSimGrid_GB-STH_1",[1,2,3]]])
9
10 #A frameCollection is a BioSimGrid data structure
11 FC = FrameCollection(datasettings)
12
13 #Get first frame
14 frame1 = FC.getNextFrame()
15
16 #Get the first frames coordinates
17 xyz = frame1.getCoordinates()
18
19 #Get the first frames metadata  (e.g. temperature)
20 temp = frame1.getFrameTemperature()
```

Example 5.5 shows how users can easily retrieve coordinate data as well as metadata about the simulation. An interface provides access to the raw trajectory data as well as the metadata, providing users with transparent access to the distributed data. This interface is the basis for all the components shown in figure 5.6. Aside from the web portal there are four key environments which are accessible to users, with the intention that these are to be used as examples for further features.

Example 5.6: Shows an example BioSimGrid deposition script. After importing the necessary modules and setting up the user's credentials, the user creates a Python list of the files containing the simulation data. In the last line the user calls the deposition module required, depending on the data format.

```
1  from BioSim.Deposit.NAMDDeposit import NAMDDeposit
2  from BioSim.Settings import UserSettings
3
4  #Setup the user credentials
5  uSettings = UserSettings('Bob')
6
7  #Python list of simulation data files
8  sim_files = {'coordinates':['file1.crd','file2.crd'],
9      'topology':'file.pdb',
10     'parameter':'file.par'}
11
12 #Envoke the NAMD deposition module
13 NAMDDeposit(uSettings, sim_files)
```

The four key areas are briefly discussed below.

- Deposition environment

  There are currently deposition scripts for all the main data formats used by existing users. Some of the different formats are discussed in section 5.5.2 and each is built upon the `Generic input` object. The `Generic input` object is intended to simplify deposition scripts, it provides all the functionality to deposit simulation data independent of its format. The format specific deposition scripts are a thin layer built upon the `Generic input` object. As support for different data formats is required, it is sufficient to write a thin plugin on top of the `Generic input` object.

  Users are provided with all the main deposition format scripts but are free to use them as a basis for custom formats. This ensures that users can deposit proprietary and future data formats into the BioSimGrid repository. Depositing data into the repository is simple and the user does not have to understand the underlying structure, this is demonstrated in example 5.6 where a user deposits data using a very simple Python script.

- Analysis environment

  BioSimGrid provides a set of over twenty analysis tools which are capable of processing and/or comparing any trajectories (or part of) within the repository.

  The source code is provided so that users can modify or create their own tools based on the scripting environment and are all written in Python. The tools are also capable of leveraging existing programs such as GROMACS (*Berendsen et al.*, 1995) and VMD (*Humphrey et al.*, 1996) using Python wrappers. This ability provides users with a simple but powerful means to integrate existing applications into the BioSimGrid tool environment.

Example 5.7: Shows an example analysis script. After importing the necessary modules and setting up the user's credentials, the user selects the data to be analysed. Users have the ability to restrict collections by frame, atom, residue and many other criteria. Once the user has selected the data it is sufficient to call the analysis tool required; there are optional parameters (*e.g.* file name shown on line 19).

```
1  from BioSim.DataRetrieval import FrameCollection, FCSettings
2  from BioSim.Analysis import AverageStructure
3  from BioSim.Settings import UserSettings
4
5  #Setup the user credentials
6  USettings = UserSettings('Bob')
7
8  #This selects frames 1,2,3 from trajectory BioSimGrid_GB-STH_1
9  datasettings = FCSettings(USettings,[['BioSimGrid_GB-STH_1'
       ,[1,2,3]]])
10
11 #This restricts the selection to only residues 10 and 11
12 datasettings.setResidueSerialNo([10,11])
13
14 #A frameCollection is a BioSimGrid data structure
15 FC = FrameCollection(datasettings)
16
17 #Performs an an analysis on the given frame collection
18 myAS = AverageStructure(FC)
19 myAS.textFilename = 'averageStructure.txt'
20 myAs.createAsText()
```

Users can easily use the existing tools to analyse their data using a script similar to that shown in example 5.7. Each of the tools has a similar interface so once users have selected the data they wish to analyse, it is possible to pass it into a number of consecutive analysis tools.

- Tools environment

  BioSimGrid provides a set of tools for the users to assist with searching and analysing the simulation metadata. The tools are capable of listing trajectories and are designed to provide users with the global identifier for trajectories that they wish to analyse. The Globally unique Identifier (GID) is used in most analysis to identify the data and can be seen in examples 5.7 and 5.8 and is described in section 5.5.2.

  Most of the other tools perform maintenance or testing but are provided as examples.

- Custom environment

  All the user scripts and tools are written in Python using the packages imported from the BioSimGrid scripting environment and are freely available under the GNU General Public License (GPL) (*GPL*, 1991). This supports and encourages user contributions and future extensibility.

Example 5.8: Shows an example of globally unique identifiers for trajectories deposited in Southampton and Oxford resources.

```
#Southampton GID Example
BioSimGrid_GB-STH_1234

#Oxford GID Example
BioSimGrid_GB-OXF_1234
```

Simulation data

BioSimGrid is a repository for biomolecular MD simulation data, each set of simulation data is called a trajectory and comprises of atom positions and velocities. This data is stored for each timestep in the simulation and also contains data about the overall simulation and atom types. A trajectory can store the positions and velocities of every atom in a molecule *e.g.* Deoxyribonucleic acid (DNA) or proteins, resulting in data in excess of 10 Gibibyte (GiB). This indicates the importance of managing the storage and shipment of the data.

There are currently many MD simulation packages, each with their own data format, *e.g.* AMBER (*Weiner and Kollman*, 1981), CHARMM (*Brooks et al.*, 1983), GROMACS and NAMD (*Phillips et al.*, 2005). All of these main formats plus a few more specialist formats are supported by BioSimGrid. The deposition environment supports further development using a well-defined Application Programming Interface (API).

Each trajectory deposited into the repository is allocated a Globally unique Identifier (GID) which is used to refer to the data in analysis scripts or between users. The GID is guaranteed to be globally unique and has been selected so as to be humanly understandable and future proof. The GID has three components each separated by an underscore, the project name, an ISO 3166-2 (*ISO*, 1995) identifier and a site unique number (see example 5.8). This format ensures that each site is independently responsible for allocating GIDs without a centralised single point of failure.

5.6 Example simulation deposition

In section 5.3 we show how an example simulation is performed by a scientist. The example demonstrates how the data files are produced and what they contain. The simulation results in the creation of a trajectory as shown in figure 5.7.

Before a user can analyse the trajectory in BioSimGrid it has to be deposited. This process imports the data into BioSimGrid making it publicly accessible through the API.

Example 5.9 shows the Python script to deposit the simulation data into BioSimGrid. The script provides BioSimGrid with the location of all the data files and an

Example 5.9: Shows the deposition script to deposit the trajectory shown in section 5.3.

```
from BioSim.Settings import UserSettings
from BioSim.Deposit.GromacsDeposit import GromacsDeposit

filenames = {
    'parameters': "md.mdp",
    'topology': "output.pdb",
    'pdb_code': "NONE",
    'name': "SJJ:  Water simulation example, 4142 H2O",
    'coordinates': ["traj.xtc"]
}

uSettings = UserSettings.UserSettings("sjj")
g = GromacsDeposit.GromacsDeposit(uSettings, filenames)
```

owner.

Once a trajectory is deposited into BioSimGrid it can be made publicly available for others to analyse.

Figure 5.7 shows the current analysis workflow and the BioSimGrid analysis workflow. Currently the trajectory data files are analysed using separate applications. Once they are deposited into BioSimGrid the same applications can be used to perform the calculations. This ensures that specialist applications can still operate on the data. BioSimGrid also provides a set of inbuilt analysis tools.

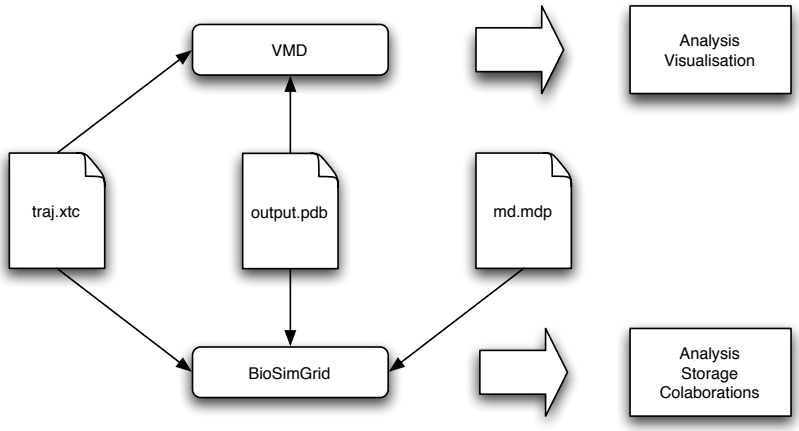

Figure 5.7: Shows the current analysis workflow and the BioSimGrid workflow. The data deposited into BioSimGrid is fully accessible by other applications.

5.7 Infrastructure

BioSimGrid integrates storage resources across many distributed locations which are geographically distant. As the project currently manages over 26TiB of data the underlying structure has to be both intelligent in retrieving data as well as robust enough to guarantee data integrity and availability.

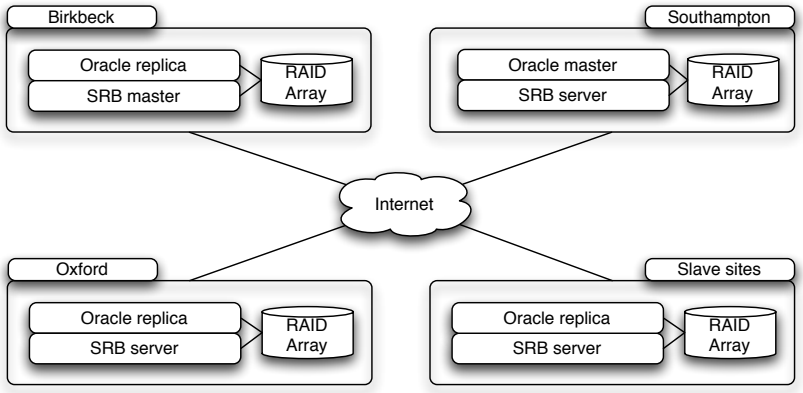

Figure 5.8: Shows the key sites supporting the BioSimGrid infrastructure. Southampton and Birkbeck run the master SRB and Oracle databases, respectively. There are many slave sites, all of which run database replicas and provide data repositories for flatfiles.

The underlying infrastructure addresses these issues by separating the data into two categories; data and metadata. Unknown to the user these are managed in two very different manners so as to utilise the most appropriate technology in each case. The metadata is managed using a Relational Database Management System (RDBMS), Oracle 10g *Oracle* (2005) which is replicated across many sites and is described in section 5.7.3. The data is managed using a distributed resource broker, SRB which is responsible for data access and replication, as described in section 5.7.1.

As BioSimGrid expands the extra nodes are added as slaves which rely on replicated data from the Oracle master, and utilise the MCAT for locating flatfiles. An overview of the SRB and Oracle infrastructure is shown in figure 5.8.

In addition to the redundancy offered by BioSimGrid all the storage arrays located at each site offer hardware and software redundancy. The hardware has a spare hot standby Hard Disk Drive (HDD) and relies on RAID 5 (*Patterson et al.*, 1988). The hardware specifications are shown in appendix A.1.

5.7.1 BioSimGrid data file formats

The largest volume of the trajectory comprises of coordinates and velocities of atoms as this information is required for every timestep. Each contains three double precision numbers for velocity and position of each atom; the X, Y and Z components. As this data is not relational by nature and is often accessed in chunks, it is not very well suited to a RDBMS; non-relational data is discussed in section 2.3.5.

As discussed in section 2.6 file systems are databases for storing and retrieving files and can be better suited to large volumes of data than databases. It is for these reasons that the coordinates and velocities are stored in flatfiles.

There are two versions of the flatfiles which are currently supported. The first

Example 5.10: Shows the information stored in the metadata file for each trajectory. This information is used for accessing the data files using the appropriate code methods.

```
{
'NumberOfAtomsPerFrame': 1855,
'FrameSize': 22260,
'DataType': 'f',
'MaxFileSize': 524288000,
'ArrayWidth': 3,
'FileNames': ['BioSimGrid_GB-STH_8.met'],
'FramesPerFile': 23552,
'DataTypeSize': 4
}
```

uses Python to serialise objects to files and the second writes out the data purely as a series of IEEE double precision floating point numbers (*Parhami*, 1999, *IEEE*, 1985). The two formats have come about as an optimisation during the development of BioSimGrid and both store the same data and support the same methods.

Both formats have the same structure for storing a trajectory. Each trajectory resides in its own directory where there is a flatfile metadata file and a series of data files. It is important to note that this is not the metadata which BioSimGrid stores about the trajectory, rather it is information about the flatfiles. The flatfile metadata file stores information about the size of frames, the number of atoms per frame, the type of data used and a list of files which contain the data (see example 5.10). This metadata file is used so that trajectory storage formats can change, but the supporting code will still be able to read the original trajectories.

Each trajectory is split into many files with a maximum size of 500 Mebibyte (MiB) for multiple reasons: i) to ensure that Operating System (OS) file size limits are not reached, ii) to speed up checksum calculations, iii) to assist with the general manageability of the data. The maximum filesize is set in the flatfile metadata file and can be changed on a per trajectory basis.

Pickled frames

This method uses the inbuilt Python serialisation method to convert Numeric arrays (*Oliphant*, 2006) to a bytestream. This bytestream is then written to a file. The serialised object can easily be re-instantiated as a Python object by reading the bytestream written from disk as shown in section 2.6.4. Figure 5.9 shows the content of a single frame as well as the content of a data file. The single frame contains all the serialised data as well as some metadata so the serialiser knows the type and size of the object when reading. The data file comprises of a series of frame objects.

Frame objects are never split across files and a file size is limited to a maximum size. Which often results in the file being slightly smaller than the maximum size. As all the frames are the same size it is easy to calculate the location of the n^{th} frame which makes access quick for any frame.

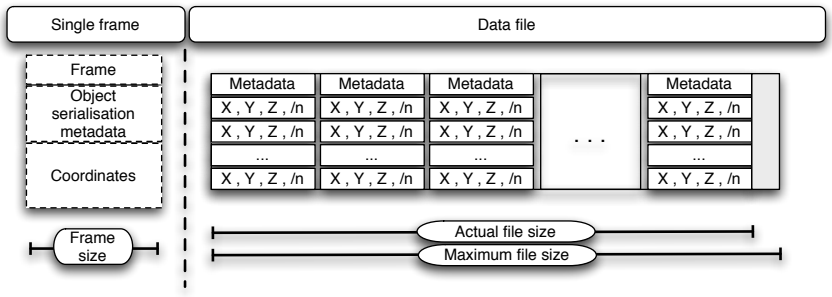

Figure 5.9: Shows the contents of a single pickled frame and shows how the frames are written in a file. Each frame comprises of metadata and coordinate tuples, ending in a delimiter.

As each frame object is serialised using the Python inbuilt functions there is no control over how this is implemented. This poses two limitations: i) there is no guarantee how it will perform in the future and ii) the implementations vary between operating systems.

For example each frame has some data stored at the beginning, essentially the object name and its size. If this data were extended in the next implementation of Python then it would not be possible to update this object as it may no longer *fit* the allocated space; without having to rewrite the entire file. Although BioSimGrid contains frame object size checks future implementations run the risk of corrupting data if a frame size increases.

In some cases users require one atom from each frame in the whole file resulting in the entire file having to be read, as each object needs to be reconstructed in memory; this is very time consuming. File methods were added to extract the required atom coordinates from within the frame object. As the object serialisation metadata is a fixed data size calculating the position in a file is relatively easy, but this breaks down between operating systems. As it is difficult to ensure that extracted coordinates are accurate the decision was taken to rewrite the flatfile data format.

Serialised frames

An alternative to using the Python pickle functions is to implement a custom serialisation method. As each file only stores objects of the same type there is no need to store the metadata for every frame. This saves space and simplifies the calculation of the coordinates for a single atom. Figure 5.10 shows how this format differs from pickled frames.

This format overcomes some of the limitations of the Python pickle method and poses an interesting example of two different data formats which implement the same file methods; this is discussed in section 5.7.2.

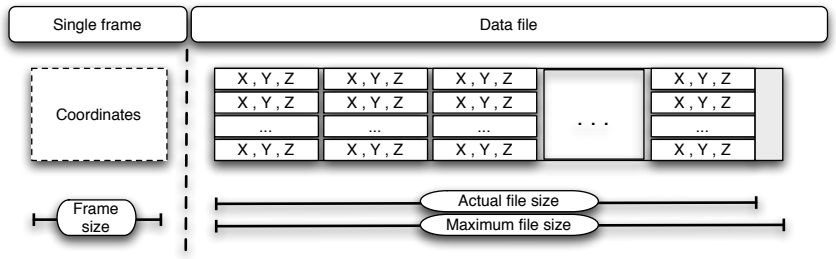

Figure 5.10: Shows the custom format used to store data in flatfiles. Each frame only contains coordinates, eliminating the need for per frame metadata. This reduces space requirements and means that single atom coordinated can easily be extracted.

Performance results

The decision to store data using flatfiles rather than using an RDBMS to manage all the data was based on a series of performance tests (*Murdock et al.*, 2004, *Ng et al.*, 2004). The tests compared the performance of the BioSimGrid coordinate data in an RDBMS, DB2 (*Mullins*, 2004) with the performance of the Python pickle approach. To complete the test an existing standard for flatfiles, netCDF (*Rew et al.*, 1997) was also compared; a summary of the results can be seen in table 5.1.

Table 5.1: Shows the performance results between two flatfile approaches and one using a RDBMS (*Ng et al.*, 2004).

	DB2	netCDF	Python Pickle
Size (GiB)	7.5	3.0	3.0
Random access (Sec)	560.8	16.4	18.6
Sequential access (Sec)	389.0	4.9	5.5

Each test shows the storage requirements and the total time for a random and sequential data access. The random access involved accessing and retrieving 1,000 frames from across a trajectory in a random order. The sequential access involved accessing the first 1,000 frames in a trajectory. The performance of the serialised frames described in section 5.7.1 is comparable to the Python pickled frames.

It is important to note that RDBMS are not slow, rather that the data is not suited to the database methodology. The database was using non-clustered indices and could potentially be optimised further. The large difference in time comes from Structured Query Language (SQL) not guaranteeing the order of data returned, thus forcing the results to be sorted in the result set.

Storage resource broker

BioSimGrid uses SRB to manage all its data files across the many distributed sites. Section 2.6.3 describes the features and capabilities offered by SRB and this section describes how BioSimGrid is implemented using SRB.

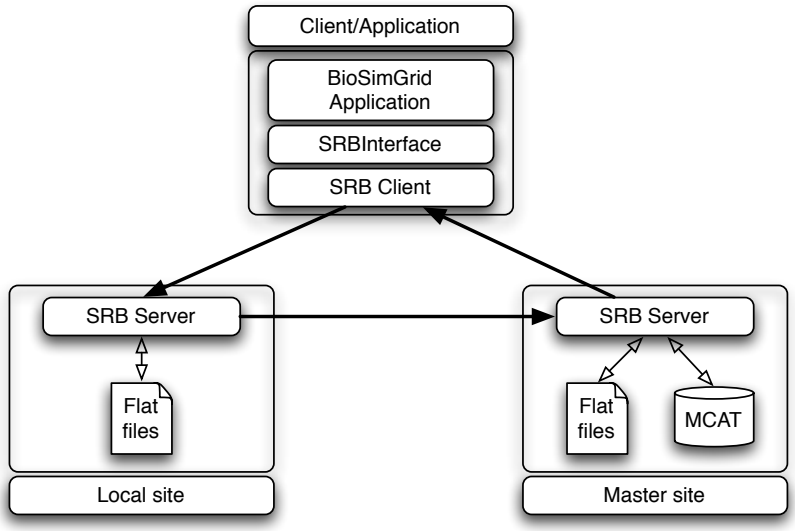

Figure 5.11: Shows how the Storage Resource Broker (SRB) manages resources. The client requests a file from the local `SRB Server`, the file is located using the MCAT and sent directly to the requesting client.

Each site has its own storage resource (≈ 4 TiB) which are managed by a single Metadata Catalog (MCAT) database currently located at Southampton. The MCAT manages all the data files in the entire BioSimGrid repository as shown in figure 5.11.

Since it is important to reduce the volume of data transported over the network each site defaults to its own resource. This results in any data deposited at a site being written to the local resource, thus reducing cross-site network traffic.

For availability and backup issues all data is replicated to at least one other site. This is managed by a daily job executed at each site. This results in data residing in many locations which helps balance load and eliminates the need to backup such large volumes of data. The replication integrity is managed using MD5 (*Rivest*, 1992, *Schneier*, 1996) hashes for each file to ensure that the data is consistent throughout the repository.

SRB is accessed using a Python interface (*Johnston*, 2005a) which provides a mechanism for accessing SRB files like local file system files. This results in the same code existing on all sites providing location transparent access to the data.

The MCAT is a single point of failure for BioSimGrid which can be seen as a limitation.

5.7.2 BioSimGrid file method implementation

Having the BioSimGrid data stored in two different file formats provides the opportunity to demonstrate the File Object Method (FOM) capabilities on a limited data set, but within a production project.

Table 5.2: Shows methods which are implemented for both the pickle and serialisation file formats described in section 5.7.1.

Method	Description
setFrame `__setitem__`	This method takes a frame in the form of a numeric array and the frame ID, which is the n^{th} frame in the trajectory. The numeric array is then stored in the flatfiles. This method is implemented using the inbuilt Python method `__setitem__` which allows users to access the object using array operations.
getFrame `__getitem__`	This method retrieves the entire contents of a frame given the frame ID. This ID is the n^{th} frame in the trajectory. This method is implemented using the inbuilt Python method `__getitem__` which allows users to access the object using array operations.
getAtoms	This method takes two parameters, a frame ID and a list of atom ID's. The data stored for the atoms whose ID's are in the atom ID list are returned. This method provides access to single and multiple atom coordinates on a per frame basis.

The first version of the flatfile database code is capable of reading and writing data to flatfiles stored locally using the Python pickle methods. The second version of the code is capable of reading and writing data to flatfiles stored within SRB (see section 5.7.1). Both versions of the code implement the same methods outlined in table 5.2 and the relevant code is shown in appendix B.1.

The two versions of the code are managed using a flatfile manager class. This is responsible for creating and returning the flatfile object. As the object returned always implements the same methods the object type is irrelevant.

The BioSimGrid code creates an instance of the database by calling a method in the flatfile manager; this returns an object which is then used to provide access to the data stored in flatfiles. The user never knows which version of the underlying code is used and is therefore oblivious to the functionality of the FOM. It provides a good working example of how the FOM can be used on a limited dataset with only a few functions.

5.7.3 BioSimGrid metadata and replication

Each trajectory stored in BioSimGrid contains metadata. This metadata stores information about the trajectory, frames and atoms as well as information about the owner and data origin. Users use this metadata to search for trajectories or frames that they wish to analyse, once these have been identified the flatfile manager is used to retrieve the actual data.

The metadata is much smaller than the data stored in the flatfiles and is very relational, thus making it suited to a RDBMS. This metadata will be queried frequently by users so needs to have a high availability. To solve the issue of perfor-

mance and availability, BioSimGrid uses database replication.

Replication is the process of maintaining multiple database instances across a distributed environment. Many databases offer replication either as a standard or additional feature. It is important to note that there is a difference between database replication and distributed databases. Replicated databases store more than one copy of a table making it available at more than one location. Meanwhile distributed databases are accessible from more than one location but a particular table is stored at only one location.

The replicated tables are often a subset of all the tables in a database (excluding site specific data and internal database data) and are often referred to as a materialised view.

The main reasons for replicating the metadata are listed below:

- Availability

  Replication increases the availability of data as it provides alternative sources and access points to the data. If one database becomes unavailable users can still access the data from any of the other sites providing good failover protection.

- Network and load balancing

  The workload can be shared amongst the different sites providing good load balancing. This assists with scaling up access to the data for large numbers of users. As there are many sources of data, users can also select the optimum server for their needs (usually the server geographically closest to them) thus reducing network costs and latency. This reduces network load dramatically which helps reduce overall data transmission time and costs. In section 2.2.4 we demonstrate the costs of transmitting large volumes of data.

- Disconnected computing

  Materialised views enable users to work on a subset of a database while disconnected from the central database server. Later, when a connection is established, users can synchronise (refresh) materialised views on demand. When users refresh materialised views, they update the central database with all of their changes, and they receive any changes that may have happened while they were disconnected.

The process of replicating data objects (tables, indices, procedures and triggers) can be implemented in many different ways depending on the application. The databases at each site fall into two categories: master sites and materialised view sites. A master site stores a complete copy of all the data objects. These are then replicated to the materialised view sites at a given point in time.

Replication modes

There are two ways to accomplish database replication, Multi-Master Replication (MMR) and Single-Master Replication (SMR).

MMR also known as peer-to-peer or n-way replication comprises many master databases, each with read and write capabilities. As a master receives an update it is then propagated to all the other masters. Oracle database servers operating as master sites in an MMR environment automatically work to converge the data of all table replicas, and ensure global transaction consistency and data integrity. Conflict resolution is independently handled at each of the master sites providing complete replicas of each replicated table at each of the master sites (*Oracle*, 2002).

SMR has one master read-write database and multiple read-only databases. Users can read from any database but can only write to a single master database. As changes are made to the master database they are replicated across to the read-only databases. Since BioSimGrid does not require disconnected databases and MMR is complicated to implement, BioSimGrid utilises SMR.

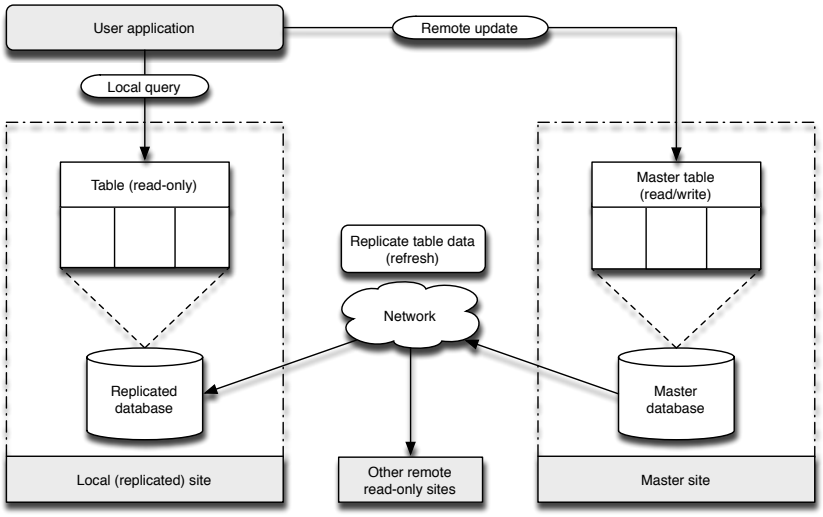

Figure 5.12: Shows how the single master replication distributes data. The user application can query the local site for data, any updates are directed to the master. The master is then responsible for pushing out the data to each of the distributes read-only sites.

When a user deposits a trajectory the deposition modules write the metadata to the single read/write master Oracle database. The bulk of the data is stored in SRB hence only the metadata is transported across the network to the master site. This works efficiently as the metadata is small. Each of the remote read-only sites distributed across the UK then poll the master for updates.

5.7.4 BioSimGrid analysis and data retrieval

BioSimGrid provides an extensible set of analysis tools which are provided as a basis for custom tools (*Arinaminpathy et al.*, 2003, *Wu et al.*, 2003). Each tool is built

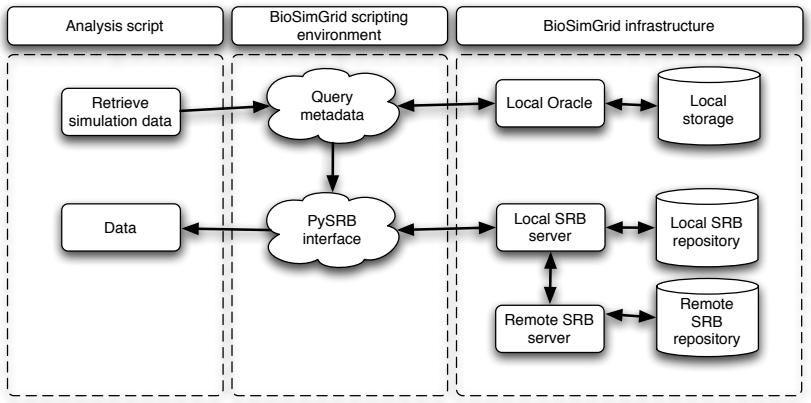

Figure 5.13: Shows how an analysis script retrieves data from the BioSimGrid repository. The metadata stored in Oracle is used to query the data files in SRB. The actual simulation data is then returned from SRB back to the analysis script.

upon the scripting environment keeping them independent of the infrastructure. There are currently over twenty tools available. Each tool accepts one or many `frame collection` objects which provide access to the required frames. The `frame collection` manages the data retrieval, providing a simple interface for accessing data.

The simulation data retrieval is shown in figure 5.13. A `frame collection` which requires frames from a trajectory, first queries the local RDBMS. This returns information about the trajectory and frames, as well as information about the part of a frame required.

The RDBMS provides information about the flatfile names and locations within SRB; this is then used to access the appropriate data. Only the data that is required and used is returned. SRB returns remote data using a peer-to-peer infrastructure resulting in efficient use of the network bandwidth.

5.8 Current user base

BioSimGrid currently has over fifty users distributed across nine locations, three sites in the United States of America and six sites in the United Kingdom. Each site has multiple TiB of storage. All the storage is linked together and managed using SRB which provides BioSimGrid with over 32 TiB of storage. Much of this storage has hardware redundancy in the form of RAID 5. This results in a usable storage capacity of $\approx$ 28 TiB.

In table 5.3 we detail the usage statistics for BioSimGrid at the time of publication. There is much ongoing work to deposit trajectories and the statistics are constantly changing. Out of the 687 trajectories stored in BioSimGrid only 91 have been published, *i.e.* are publicly accessible.

Table 5.3: Shows the statistics for all published and unpublished trajectories stored in BioSimGrid.

Total number of trajectories	687
Number of released trajectories	91
Total simulation time (ns)	42594065
Average simulation time (ns)	62000
Average frames per trajectory	8168
Average residues per trajectory	5911
Average atom count (inc solvent)	20979

Table 5.4: Shows the number of published trajectories in each category.

Number	Category
40	scorpion toxin
33	outer-membrane protein
9	potassium channel
4	membrane protein (other)
3	amino-acid binding protein
3	acetylcholinesterase
2	calcium channel
1	glutamate receptor
1	lipid bilayer
1	Lysozyme

In table 5.4 we show the published trajectory categories. This provides an insight into the different chemistry areas where BioSimGrid is proving useful.

5.9 Discussion

Using the FOM within the BioSimGrid project tests the concept in a limited and easily managed environment. Not all the FOM features are implemented in this version as it runs on a production environment where data integrity is crucial. Those features implemented are discussed below.

- Shell method execution

  The flatfile manager can be executed using Python, which enables the data to be queried and returned. It is the user's responsibility to format the data returned and this feature is not intended for use by the BioSimGrid users. The feature was intended as an example of the FOM feature but has subsequently been useful for testing results returned from the data queried.

  The command line features are used to examine data and demonstrate the usefulness of such a feature.

- Code quality assurance

Monitoring the code submitted by users is very difficult and assessing the reliability and accuracy of such code is nearly impossible. The BioSimGrid is probably the closest the FOM will get to having quality assurance on code as users are not free to deposit code. This is seen as a disadvantage as the aim of the FOM is to support code collaborations. It does provide the opportunity to test and examine the scripts and provide a measure of quality.

As the scripts are used frequently and have limited functionality they are easy to test and any bugs are quickly reported.

- Method execution – results as objects

  Each of the methods shown in table 5.2 returns data as Python objects. These objects are then passed to the calling code. As Python is a dynamically typed language execution is simple, the calling function is provided with an object which it can use (*Lundh*, 2001b).

- User code submission – language dependent

  There is no mechanism for users to submit code but there are two different file types supported. The code was added in at the administrator level as the data storage layer of BioSimGrid is transparent to users.

  This provides a controlled environment with which to introduce 'user code' and test the capabilities for two different scenarios. The BioSimGrid project supports further code submissions.

- Code association – file methods

  Each of the methods in table 5.2 is associated with the file containing the data.

- Code association – file type methods

  As there are only two file types; those in SRB and those on the local file system. The filename does not determine which code is invoked, rather it is the location. This has come about as the file names and types are the same but the code for managing each file depends on its locations.

5.9.1 Limitations

Applying the FOM to BioSimGrid highlighted some of the limitations of both the FOM and the BioSimGrid project. These limitations are now discussed and are used as the basis for changes to the FOM described in section 6.

Remote execution

As the data is distributed across many sites the next logical progression is to execute the methods on the machine where the data resides. Shipping the processing closer

to the data is the most economical mechanism to extract data, especially where the data returned is smaller than the data required to produce the results. Using the FOM on a file remotely provides an excellent mechanism to load balance and reduce bandwidth. SRB provides a command to remotely execute a shell script on the machine where the data exists. The standard output and standard error (*Rosen et al.*, 1999) are then piped back to the calling machine.

In the current version of SRB used by BioSimGrid this feature is not fully developed and has some serious limitations, which make it unusable.

If this feature worked as expected the data which can be returned is limited to text and shell scripts. This is not acceptable for the FOM and one of the advantages is that users can retrieve objects compatible with their current programming language.

This limitation indicates that although the concept is valid, the technology used is inappropriate. As this feature was not in the original FOM discussed in chapter 3 a proposed change to the FOM is described in chapter 6 resulting in a change of technology.

Network traffic

One limitation of BioSimGrid is the inefficiency of the transport of data. When simulation results are generated they are stored locally, these are then processed using the deposition code and directly written into the BioSimGrid database, resulting in the data residing at two locations.

Additionally as the simulations are not processed on the BioSimGrid machines and are copied to a BioSimGrid node there are normally three copies of the simulation data in up to three different locations (excluding any SRB replication). As each simulation dataset is potentially large this is an inefficient use of resources.

Repetition

BioSimGrid has a core set of analysis tools described in section 5.7.4. As there are only a few tools it is not inconceivable that users might run the same or similar analysis scripts. One of the main disadvantages of the BioSimGrid project is that users cannot see what analysis has already been processed and they do not have access to any analysis results from other users.

As analysis is computationally expensive, it is more efficient to save the results and make these available to other users. Section 6.3.5 describes how the FOM can be adapted to meet this need.

5.10 Summary

In this chapter we have introduced the BioSimGrid project and described the motivation behind the project. BioSimGrid provides a repository for scientists to deposit simulation data in different formats making the data accessible to users.

The data within BioSimGrid can exist in two different formats, those stored in flatfiles locally and those stored in SRB flatfiles. These two formats have come about as part of the evolution of the BioSimGrid project and provides a controlled environment to implement the FOM.

Using the FOM to implement a number of BioSimGrid functions highlights the benefits and limitations of the model. The main limitations are the inability to work in a distributed environment, inefficiency of network and resource usage and the repetition of calculations.

Whilst these limitations do not affect the functionality of the FOM they are fundamental to its overall usability and efficiency aims. These limitations provide a basis for modifying and adding features to the FOM and are discussed in chapter 6.

Chapter 6

Method adaptation

6.1 Introduction

In chapter 3 we proposed the File Object Method (FOM) concept and outlined its features and capabilities. The FOM has been explored using two different implementations: the prototype shown in chapter 4 and a production project, BioSimGrid, shown in chapter 5.

Both these implementations have provided an opportunity to examine how the FOM performs on data in both test and production environments. The next logical progression for the FOM is to look at the findings from these projects to see how the method can be improved. Both implementations have provided valuable feedback with which to recommend FOM modifications.

This chapter provides a list of enhanced features and capabilities of the FOM aimed at meeting the objectives outlined in chapter 1.

6.2 Motivation

The prototype shown in chapter 4 provides an implementation of the FOM. This implementation runs on a single machine and enhances the user's data management capabilities. For the FOM to meet the objectives set out in chapter 1 some changes are required.

The FODB prototype does not fully support access to data across a distributed infrastructure but supports the submission of user code. The BioSimGrid project manages data across a distributed infrastructure but is very restrictive about submitting user code.

In this chapter we look at the advantages of both the FODB and the BioSimGrid infrastructures in order to enhance the implementation-independent FOM specification. Using this we describe additional features which future FOM implementations should incorporate to enhance the user's data management capabilities.

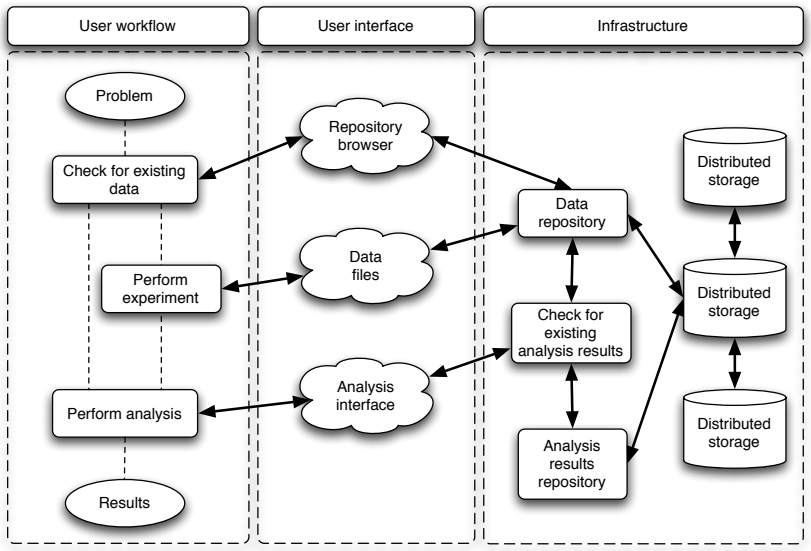

Figure 6.1: Shows the proposed changes to the FOM workflow. The FOM checks the analysis repository to see if a FOM method has been executed previously and returns the cached results if they exist.

6.2.1 Proposed workflow

In section 2.2 on page 5 we took the existing user's workflow and proposed an enhanced workflow based on the FOM as shown in figure 3.1 on page 43.

The FOM proposes moving the analysis scripts from the user's space into the infrastructure to facilitate collaborations. Figure 3.1 shows how the proposed FOM keeps the user's workflow consistent but assists with managing the data. The FOM changes the underlying infrastructure and is implemented in the prototype described in chapter 4.

This is the basis of the BioSimGrid workflow shown in figure 5.5 on page 79, which supports distributed storage. It encourages users to check for existing simulation data before performing a simulation.

We now propose to enhance the user's workflow further by including a cache of previously executed analysis operations, as shown in figure 6.1. The differences appear in the infrastructure as the data is no longer stored on a local file system.

All the experimental data is deposited into a repository along with the results from any analysis. When a FOM method is called the results are returned to the user as well as stored in the FOM infrastructure. If the same method is executed again, the FOM first checks the cache thus eliminating the recalculation of results. If the underlying data changes, the cache is cleared of all related analysis results.

The *perform experiment* and *perform analysis* stages in the workflow remain unchanged permitting users to continue with their existing workflow unhindered.

6.3 Method modifications

Using experiences gained from the prototype shown in chapter 4 and the production project BioSimGrid shown in chapter 5, we propose modifications to the FOM described in chapter 3. These experiences have suggested additional features which are appropriate to add to the FOM and can be used to support the workflow described in section 6.2.1.

The proposal is to adapt the FOM to work across multiple machines transparently to the user. The key motivation is to assist with collaborations and to accelerate the exchange and sharing of code. An implementation-independent description of these features is described below.

6.3.1 Data distribution

Distributing data across multiple machines and managing access, replication and data integrity is a problem in itself. Data transmission and nodes can be unreliable and any distributed environment has to be able to manage anomalies not found in a standalone environment (*Anderson*, 2001). This feature enables the FOM to work over a distributed environment so that data and code can easily be exchanged between users. The FOM is not intended to implement a distributed environment, rather it should be able to run on top of a distributed environment.

Each file cannot be split between resources as this would result in the user's code having to manage this, unless the distributed layer makes it transparent. All the files available across the distributed FOM have to be visible to the user as if they were locally stored. The users should not be able to distinguish if they are working on a distributed repository or a local repository.

Each data file can exist on more that one location, resulting in redundant backup copies which improve availability.

6.3.2 Remote method execution

As the data is to be distributed then the method execution has to be able to operate remotely. This is more difficult for some features than others but the method has to execute and return results as if it were run locally.

6.3.3 Load balancing

As a data file can exist on more than one resource there is an opportunity to load balance method execution. This feature enables the FOM to select the most appropriate machine to execute a method based on the machine load.

6.3.4 Security

Security is an issue which should be included in any software design (*Cwalina*, 2005). The prototype shown in chapter 4 supports the standard file system permissions but as the data becomes distributed across sites there is a need to provide assurances about the security of remote data as well as the transmission method. The BioSimGrid project provides an insight into securing distributed data across a network but does not offer any code execution security.

The FOM does not implement any security functionality as it is intended to leverage existing technologies. As with most distributed systems if the physical machine is compromised then it is assumed that the FOM security is also compromised. Below we show how the FOM secures the data, code and transport layers.

Data

As all the data is stored in files, the permissions are the responsibility of the operating system. The FOM has to be able to support data access based on the operating system permissions. This ensures that the data is secured using a mechanism with which the users are familiar.

Code execution

The FOM enables users to submit code which is then executed on the machine holding the data. Access to the data files are restricted at the operating system level, ensuring the data is safe. For the FOM to be successful the code execution has to be managed to ensure that it is not possible for malicious code to damage data or disrupt services. This FOM feature is required to ensure that the user's code runs in an isolated environment such as a sandbox (*Gong et al.*, 1997).

Transport security

The FOM has to guarantee that all data transmitted between machines is secured and encrypted. The security is not performed by the FOM but by leveraging existing technologies. For example data communication can be secured using Secure Sockets Layer (SSL) or Internet Protocol Security (IPSec) , both of which are widely accepted across the technological community (*Tiller*, 2000).

6.3.5 Method results cache

Currently BioSimGrid users are unable to see which analyses have already been calculated by other users and rerunning an analysis wastes resources. The cache can compromise security as one user *could* work out what another is analysing.

When a FOM method is executed the results have to remain in a cache. This cache serves as an automatic repository for analysis results which reduces recalculation times.

This has implications for the users as often they wish to keep their work private, at least until published. In the FOM we propose to keep the cache running at all times, but not making it possible for users to browse other users' analyses.

For example if a user executes a method previously executed by another user the results will be retrieved from the cache. A user has no mechanism to list which methods have been executed previously by other users. The time taken for a calculation to complete may provide users with an indication that the calculation has previously been computed.

6.4 The .NET framework

In chapter 4, Python was selected as an appropriate language to implement the first FOM prototype. As the FOM features have become more defined and the features more demanding, there is a need to review the existing technologies.

The FOM features require that future implementations are both secure and capable of managing distributed data. To ensure that the FOM is successful it has to support many languages.

It is for these reasons that the .NET framework (*Richter*, 2002) is considered for the final FOM implementation.

The .NET framework is the next evolution of the Component Object Model (COM) which has been extensively utilised in the past (*Gordon*, 2002). There are various variants of COM, mainly Object Linking and Embedding (OLE), Distributed Component Object Model (DCOM) and ActiveX (*Barry*, 2003). The .NET framework provides language independent interoperability between existing libraries and promotes the implementation of secure code.

It is similar to the Java 2 Platform Enterprise Edition (J2EE), developed by Sun Microsystems which supports multi-platform applications (*Armstrong et al.*, 2005). .NET has limited platform support as discussed in section 6.4.1.

The .NET framework supports over 23 programming languages and supports many third-party languages. The ability to extensively support most programming languages is due to the Common Language Infrastructure (CLI), described in section 6.4.1. It is this support for the most common programming languages that makes the .NET framework an attractive platform for the FOM.

6.4.1 Common language infrastructure

The CLI is the key .NET framework design feature, which provides support for multiple programming languages. It is also responsible for exception handling,

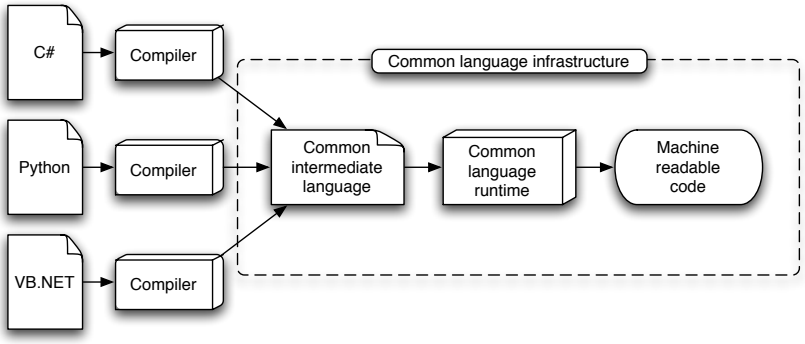

Figure 6.2: Shows the Common Language Infrastructure implemented in the .NET framework.

resource allocation and garbage collection (*Jones*, 1996). The .NET implementation of the CLI is called the Common Language Runtime (CLR), which is similar to the Java Virtual Machine (JVM). The JVM takes the platform independent byte code generated by the Java compiler and executes it by interpretation or Just-in-Time (JIT) compilation.

Each programming language supported by the .NET framework has its own compiler which produces platform-independent code, called the Common Intermediate Language (CIL). The CIL ensures that any .NET compiled code can run on the CLR. The CLR is platform-specific and it compiles the CIL into machine-specific code as shown in figure 6.2.

6.4.2 Message transmission optimisation mechanism

The .NET framework supports remote code execution through web services and .NET remoting (*McLean*, 2002).

Web services use Extensible Markup Language (XML) to pass messages between clients. This is inefficient when transferring binary data. It is for these reasons that new standards have been developed and are discussed below.

Message Transmission Optimisation Mechanism (MTOM) is a standard which has evolved to address the issue of efficiency and interoperability of XML message transmission. The MTOM encoding of a document looks similar to the standard XML document serialisation, but optimises large blocks of binary data by transmitting them as binary attachments (*W3C*, 2005a).

MTOM is intended to replace the existing standards for attachments, Direct Internet Message Encapsulation (DIME) (*Microsoft*, 2002a,b), WS-Attachments (*Microsoft*, 2002c,e) and SOAP with Attachments (SwA) (*Microsoft*, 2005b) as these have limitations.

For example, SwA uses strings to denote the beginning and end of binary data. This results in the entire dataset having to be searched to locate the end. SwA cannot be represented in an XML infoset, unlike MTOM. This causes problems with

web service model and results in the inability to secure the data using standards such as WS-Security (*Microsoft*, 2002d). DIME is more efficient than SwA, but cannot be represented as an XML infoset.

One of the main advantages of MTOM is that it preserves the XML formatting of a document ensuring that it is possible to combine MTOM messages with WS-Security (*Microsoft*, 2002d) so that SOAP *W3C* (2003) messages can be signed and encrypted even if they contain binary attachments.

MTOM leverages the XML-binary Optimised Packaging (XOP) standard which provides a mechanism for serialising an XML infoset into an `XOP Package`, which is an abstraction in the specification (*W3C*, 2005b). One implementation is to use Multipurpose Internet Mail Extensions (MIME) Multipart/Relate to attach the binary data (*Levinson*, 1998).

At the time of writing MTOM is implemented in a few publicly-available beta toolkits, the key implementations are:

- Indigo is a set of .NET (*Chappell*, 2005) technologies for building and running connected systems. It is intended as a communications infrastructure built around the Web Services architecture shown in section 2.8.1. The existing beta version currently implements MTOM.

- Web Services Enhancements (WSE) version 3.0 is currently released as a Community Technology Preview (CTP) and implements MTOM. It is a .NET add-on which provides the latest web service capabilities.

- Java API for XML Web Services (JAX-WS) version 2.0, currently released as an early access product for Java (*Chinnici et al.*, 2006) developers.

In addition to MTOM there are other web service standards which are appropriate for transmitting messages (data) between clients. To ensure reliable message transmission we propose the use of WS-ReliableMessaging (WS-RM) (*Bilorusets et al.*, 2005).

Reliable messages are often not sufficient, for example as with databases, users often want operations to be transactional. Either all messages arrive and are processed or the system remains in the state before messages were sent. The WS-Coordination (*Cabrera et al.*, 2005a) framework provides support for protocols that coordinate the actions of distributed applications. For example, for short duration, ACID transactions it supports WS-AtomicTransaction (*Cabrera et al.*, 2005b) and for longer running transactions it supports WS-BusinessActivity (*Cabrera et al.*, 2005c).

The messages transmitted between clients may contain sensitive information. WS-Security (*Atkinson et al.*, 2002) provides a mechanism for securely transmitting these messages.

6.4.3 Security

The .NET framework has mechanisms to ensure that malicious code is either not executed or has restricted operations.

When a .NET assembly is executed the CLR first obtains evidence about the assembly, using its associated metadata. This is used to identify the code group to which the assembly belongs. Each code group has a set of permissions determined by the machine administrator. When the CLR detects an unauthorised operation, it throws a security exception and halts execution.

Code downloaded from an untrusted source, *e.g.* the Internet, is automatically stored in a sandbox (*Gong et al.*, 1997).

The .NET framework provides users with the ability to digitally sign their code. Code that is signed is said to have a Strong Name (SN)(*Jones*, 2003). This ensures the authenticity and integrity of users' code and ensures that it has a globally unique name which helps prevent spoofing.

The .NET framework supports impersonation, which permits the execution of code as if run by a different user. This is beneficial to the FOM where users' code needs to be executed with restricted capabilities.

6.4.4 Managed code

Managed code is a feature of the .NET framework which makes the CLR responsible for the managed execution of user code. As shown in figure 6.2, the CIL is executed by the CLR. This provides the CLR with the ability to determine what operations the user's code will perform. When the CIL is compiled the compiler inserts garbage collection hooks (*Jones*, 1996), array bounds, indices and other checks to improve performance and security. The CLR can halt execution of a process if it attempts to perform an illegal operation such as memory or file access.

This ability to ensure that malicious code cannot be executed is an attractive option for the FOM.

6.4.5 Alternative implementations

The CLI has more than one implementation, of which the CLR is the most advanced. The other main implementations are aimed at porting the .NET framework capabilities to other platforms. The three most popular are discussed below:

- The Shared Source Common Language Infrastructure (SSCLI), codenamed Rotor is Microsoft's shared source implementation of the CLI and is not intended for commercial purposes. The first version comes pre-configured to run on Windows, FreeBSD (*McKusick*, 2004) and Mac OS X (*Apple*, 2006). The most recent version, `v2.0` currently implements most of the .NET framework classes.

- The DotGNU Portable .NET project implements the CLI for many different architectures and platforms (*DotGNU*, 2006). It only supports two programming languages, *C* and *C#*.
- Mono is very similar to the SSCLI implementation and is licensed as Open Source Software (OSS) promoting contributions from interested parties (*Mamone*, 2005).

6.5 Active directory

Many users can submit code to the FOM and they may want to restrict user access to methods, code and data files. The FOM does not aim to implement any security features, but prefers to leverage on existing technologies. All the access rights and user management are controlled by the host operating system. This may not provide a suitable security granularity. For example, it is difficult to restrict a single FOM method to a user or group of users. Enterprise wide user management systems already exist and may be suitable for future FOM implementations.

One existing, widely adopted mechanism for managing users across an enterprise infrastructure is Active Directory (*Spealman et al.*, 2003). Active directory provides system administrators the ability to centrally manage many machines using a role based policy.

These policies can be applied to resources (Storage) , services (FOM methods) and users, thus providing a fine level of control.

Active directory requires a large infrastructure, *i.e.* a dedicated domain controller. For small FOM implementations this may be too large an overhead, an alternative such as Active Directory Application Mode (ADAM) may be more appropriate. On a standalone machine, *i.e.* not belonging to a domain, it is possible to apply local policy to restrict users to certain applications or data.

6.6 Discussion

The FOM is intended to promote collaborations across a wide variety of users. It is for this reason that the FOM has to support distributed data resources. As with BioSimGrid, users must have multiple entry points to the data. This hardens the system in the event of failure but also assists with load balancing. If the data is replicated, the analysis can be performed at multiple locations simultaneously. It is important that the processing is moved to the data to reduce network load. Storing the data in more than one location can also assist in making the data accessible to more users and assists with data persistence.

In the BioSimGrid implementation it is common for a user to copy data files many times, once from the machine where the data is created to the BioSimGrid

node. From here it is parsed and deposited thus copying it again into an internal format. As the data gets large, copying becomes an expensive inconvenience. It is therefore important that the FOM supports the direct copying or streaming of data into the repository.

The FOM analysis results cache provides an opportunity to optimise the analysis of data. There are several implications with storing the analysis results. The cache can get too large, as with simulations and experiments, where the data is large it *may* be quicker to recalculate than to store the data. The FOM also has to monitor the data files; when a file is changed the cache also has to be purged. Since the cache is transparent to the users it can greatly enhance the users capabilities. The cache size and the lifespan of data are left as implementation specific variables.

We discussed the capabilities of the .NET framework and highlighted the capabilities which are beneficial to any FOM implementation. The main advantages are the advanced security features and the support for multiple languages. The main drawback is the limitation of supported platforms. A FOM implementation running on the .NET framework is restricted to the Windows platform. In section 6.4.5 we discussed some alternative CLI implementations which demonstrate ongoing work to support additional platforms.

6.7 Summary

In this chapter we recommend modifications to the initial FOM described in chapter 1. These modifications are based on findings from the two FOM implementations, the FODB and the BioSimGrid project.

The modifications include distributing the FOM repository across multiple resources to provide better data accessibility and redundancy. We propose adding a FOM caching repository to store all previously calculated analysis results. This eliminates repeat calculations and optimises the user's experience.

We discuss the key features of the .NET framework and some alternative implementations which can benefit future implementations of the FOM.

Chapter 7

Final method implementation

7.1 Introduction

In this chapter we take the FOM concept shown in chapter 3 and use the recommendations in chapter 6 to produce a final proof of concept prototype. This prototype, called the Storage and Processing Framework (SPF) demonstrates all the FOM features in a secure and distributed environment.

The SPF is implemented using the .NET framework and is based on a Service-Oriented Architecture (SOA). The underlying infrastructure supports a secure and distributed file system upon which we demonstrate the FOM features using two examples. These examples are written in different .NET languages and used to demonstrate the multi-language support of the SPF.

In this chapter we outline the infrastructure and capabilities of the SPF and in chapter 8 we compare this prototype with previous implementations.

7.2 Overview

The SPF is a .NET implementation of all the FOM features (*Johnston et al.*, 2006a). It has three key components: the storage service, the storage manager and the client layer as shown in figure 7.1.

The storage service manages the data, mapping it to a physical resource. The SPF can have many storage services each controlling a single resource. All the storage services are controlled by a single storage manager. The storage manager is the point of entry for the client layer which exposes all the SPF features to the end users. As each storage service has to register with the storage manager, the user layer can locate any data in the SPF.

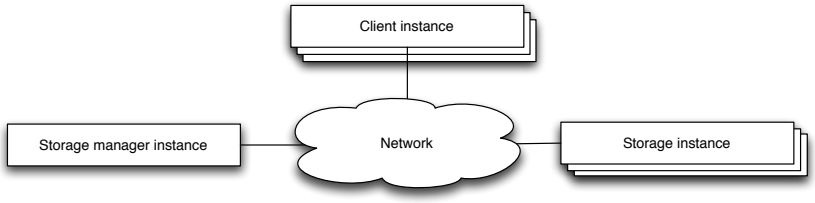

Figure 7.1: Shows the three key components of the SPF. Each component runs as a web service providing interoperability across a distributed environment. A single storage manager supports many client and storage instances.

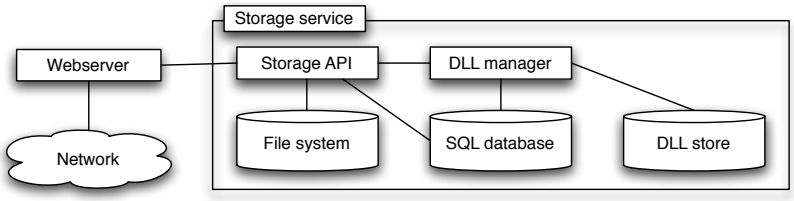

Figure 7.2: Shows the key components of the storage service. The storage API methods are all exposed through a web service.

7.3 Storage layer

The key objective of the storage layer is to provide a mechanism for accessing files on a given machine via a web service. By implementing the storage layer it is possible to show how this File Object Method (FOM) model can be used to perform calculations in a distributed environment. This storage layer is not intended to be a substitute for a distributed file system and is merely a testing platform for the SPF.

The storage layer is responsible for making the files transparently accessible to the client layer, regardless of location. The storage layer is built up with many storage services and a single storage manager, both of which are web services.

7.3.1 Storage service

Each machine has one storage service which manages the data stored on that resource. The storage service is responsible for taking files and storing them on the storage space provided by a resource. The service then responds to requests for files and information about files.

The storage service instance has two key components: i) the storage API and ii) the Dynamically Linked Library (DLL)manager, as shown in figure 7.2.

The storage API maps the SPF file requests to local files and is responsible for invoking the DLL manager. When a file is deposited, it is stored in the local file system and the name and file type are registered with the local SQL database.

The users .NET code is compiled into an assembly called a DLL. The DLL manager stores all the user's code and maps it to files and file types. When a file is

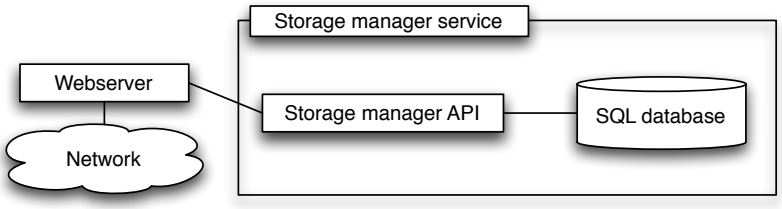

Figure 7.3: Shows the key components of the storage manager. The locations of all known storage services and a cache of known file locations are all stored in an SQL database.

selected the DLL manager provides information about the user's code associated with that file and what methods are available. When a user executes a method the DLL manager locates the code and executes the constructor using the local file as a parameter. The results are returned back to the user through the storage API.

The DLL manager caches the results returned by an SPF method. These results are stored in the DLL SQL database and are used to speed up SPF execution. The DLL cache stores the last modified time of all SPF files. When an SPF method is invoked the timestamps are compared. If an SPF file has been altered the SPF cache is flushed.

7.3.2 Storage manager

Each storage service has to register itself with the storage manager which is responsible for receiving file requests from the client layer and returning a list of storage web services that store the requested data file. The storage manager stores the storage service metadata in a SQL database as show in figure 7.3.

There are many storage services, one per machine and one storage manager in the SPF. The FOM architecture can support more than one storage manager to allow users to have more than a single point of entry. This can assist with load balancing although the SPF implementation utilises a single storage manager.

The storage manager is not responsible for managing the locations of files or for managing any metadata about the files. Its function is to provide a point of entry for the client service. When a client requests a file it first asks the storage manager to return the file providing the location of the client web service which requires the file. The storage manager is a lightweight index of the files stored in the SPF; it does not fully index all the files stored in the SPF. If a requested file is unknown the storage manager polls all the known storage services requesting the file. The location of files are cached in the storage manager and the client layer. This simple mechanism provides a framework upon which it is possible to test the FOM.

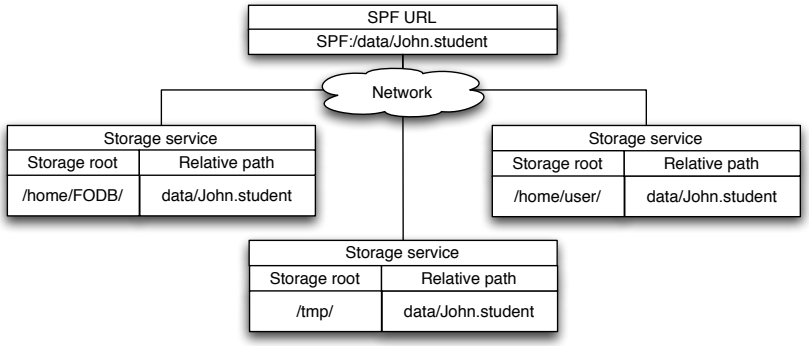

Figure 7.4: Shows how a single SPF URI maps to more than one replicated file. Each file is stored in a separate storage service and the SPF URI is relative to the storage service root directory.

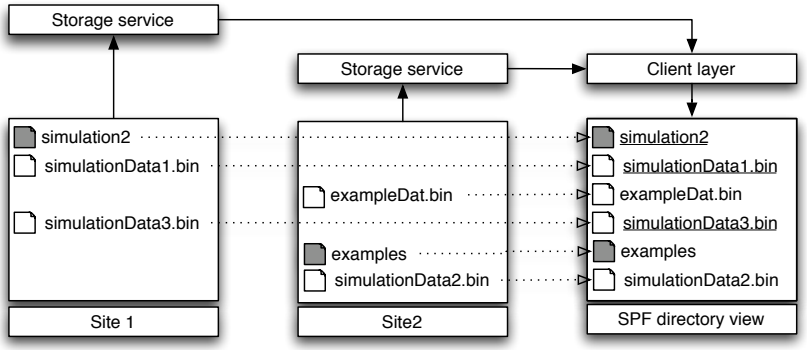

Figure 7.5: Shows how the files distributed across two storage services appear to the client layer. Files with the same SPF URI are taken as replicas of the same file.

7.3.3 File objects

The storage layer identifies files using a SPF Uniform Resource Indicator (URI) as shown in figure 7.4. The URI identifies a file to the end user which may reside in many locations. In figure 7.4 the file `John.student` is stored in three locations. Each storage service has its own root folder where all the SPF data files are located. The SPF URI is relative to the root folder. Thus each file does not need to have the same location on each storage service. The absolute filename is retrieved by replacing the '`SPF:`' prefix with the storage service root folder name.

The client layer treats a replicated file as a single file object. When a user requests data from the file, the most appropriate storage service is selected. This is based on Central Processing Unit (CPU) load but can be substituted for other machine parameters. If the file replications are not synchronised, the client layer flags the file as dirty. If the user chooses to ignore this then the file with the most recent time stamp is used. When the client layer marks a file as dirty it notifies all the out-of-date replicas of the storage service where the most recent data is stored. The replica storage services then synchronises the data files.

The client layer can query all known storage services for a list of files and direc-

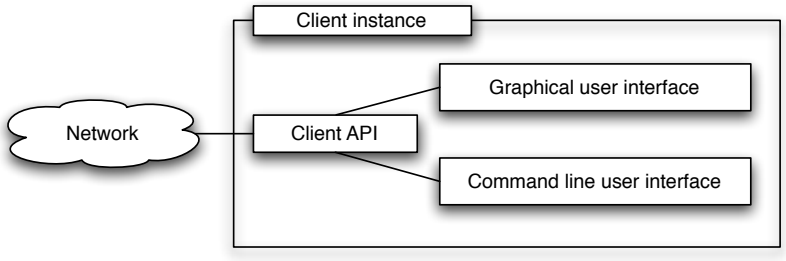

Figure 7.6: Shows a single client instance. The client API provides a complete interface to the SPF across a network. The API currently supports a command line and graphical user interface and is extensible for future applications.

tories contained within a specific SPF folder. Figure 7.5 shows how the data from two storage services is merged to produce a client view.

7.4 Client layer

All the client features are accessible through the client Application Programming Interface (API). This interface provides a .NET library upon which all the client layers are built. Currently two layers have been implemented: i) the command line user interface and ii) the Graphical User Interface (GUI) as shown in figure 7.6.

The client API must have a point of entry into the SPF. This is accomplished by supplying the location of one storage manager web service. Once this is established, the API can query the SPF for files, names, locations and associated user code. If the API does not know of the storage service where a data file is located, it first asks the storage manager to supply a list of valid storage services. The locations of files are cached in the API to speed up frequent access. The API can then query one of the storage services to retrieve information about the associated code.

The client API provides methods to retrieve data files as well as query the associated user code. This provides a list of associated user classes which can in turn provide a list of associated user methods.

Once the user's method has been selected the API can call the storage service to invoke the associated code on the selected SPF file. The results are then transferred back to the API layer where users are free to manipulate the data. The results of a user method invocation are returned as .NET objects which the users can then use programmatically in future code.

The GUI provides a visual representation of the underlying API capabilities and is intended as an example application of the SPF. In section 7.5 we demonstrate the GUI capabilities using the examples show in section 7.6. All the features exposed in the GUI can be programmatically utilised by the user.

Example 7.1: Shows an example C# class with methods to provide information about a file.

```
using System;

namespace AdvancedFileInfo
{
    public class AdvancedInfo
    {

        public AdvancedInfo(string FileName)
        {
           //Constructor
           getInfo()
           ...
        }

        public string contact(){
            return this.ownerContact;
        }

        public string getFullLocalPath(){
            return this.fileInfo.Name;
        }

        public System.DateTime getLastAccessTime()
        {
            return this.fileInfo.LastAccessTime;
        }
    }
}
```

7.5 Example user code

To demonstrate the FOM capabilities we provide two example pieces of user code. These are used in sections 7.6 and 7.7 to demonstrate the user's workflow and FOM features within the SPF.

The first example consists of a C# class which provides metadata about a specific file as shown in example 7.1. The `AdvancedFileInfo` class takes a file name as a constructor parameter and provides three methods to return information about the selected file. Each of the methods are described below:

- The `contact` method returns a string indicating whom the user can contact for further information about the class.
- The `getFullLocalPath` method returns the full path and file name of the selected file. This is useful in a distributed environment as the SPF file name masks the real location of the data files.
- The `getLastAccessTime` method returns a `DateTime` object showing when the file was last accessed.

The second example provides information about text files. It is designed to operate on any text file and is written in Visual Basic as shown in example 7.2.

Example 7.2: Shows a visual basic class which provides information about text files.

```
Public Class WordCount

    Public Sub New(ByRef filename As String)
        ...
    End Sub

    Public Function countWords()
        Return txt.words
    End Function

    Public Function countLines()
        Return txt.lines
    End Function

    Public Function countCharacters()
        Return txt.chars
    End Function

End Class
```

The `WordCount` class takes a filename as a constructor parameter and provides the following methods to return information about the selected file:

- The `countWords` method returns the number of words in the selected text file.
- The `countLines` method returns the number of lines in the selected text file.
- The `countCharacters` method returns the number of characters in the selected text file

Both .NET classes are compiled into a .NET library (DLL) which are then imported into the SPF so they become available to all users.

7.6 Client layer workflow

In this section we describe the user's SPF experience using the client GUI interface described in section 7.4 and the examples provided in section 7.5.

In this scenario we have two storage services running on machines called *Formido* and *Onerous*. Examples 7.3 and 7.4 show the files stored in the storage service. The two student data files are stored on separate machines but the SPF file view merges the directories to appear as a single directory. The `readme.txt` file appears on both machines and will appear as a replicated file.

Both of the example .NET libraries shown in section 7.5 have been added to the SPF. The `AdvancedFileInfo` DLL is associated with all files and the `TextInfo` DLL is associated with any file with the `txt` extension.

Example 7.3: Shows the files stored in the storage service running on *Formido*.

```
sjj@formido> tree .
.
|-- StudentData
|   `-- Fred.student
`-- readme.txt
1 directory 2 files
```

Example 7.4: Shows the files stored in the storage service running on *Onerous*.

```
sjj@onerous> tree .
.
|-- StudentData
|   `-- John.student
`-- readme.txt
1 directory 2 files
```

7.6.1 Selecting an SPF method

The SPF windows client is shown in figure 7.7 and is divided into five regions. The lower region displays the properties of the currently selected object. In figure 7.7 this region displays information about the machine *Formido*.

The upper four regions are used to find and select methods associated with data files. For example, to select the `GetLastAccessedTime` method on the `readme.txt` file, the user's four steps are shown in figure 7.7.

- Step 1

  When the user interface loads, the client layer connects to a storage manager web service. The client then requests that all storage services return a list of all files and folders stored within the root SPF directory. These files are then collated and displayed in the *SPF file* tree view of the user interface. In this example the user has selected the `readme.txt` file.

- Step 2

  When a file is selected in the *SPF file* tree, the *packages* tree is then populated. This provides a list of all the packages (collections of classes) which are associated with the selected file. In this example the `readme.txt` file has two associated packages, the `AdvancedFileInfo` and the `TextInfo` package. Not all storage services have the capability to run all user code as the DLL may not exist on a particular resource. The client layer will automatically select a storage service to execute a method base on the machine's load. In the GUI a user can see which machines are capable of running particular user code. Both the associated packages can run on either machine (*Onerous* or *Formido*). In this example the user has selected the `AdvancedFileInfo` package on the machine *Formido*.

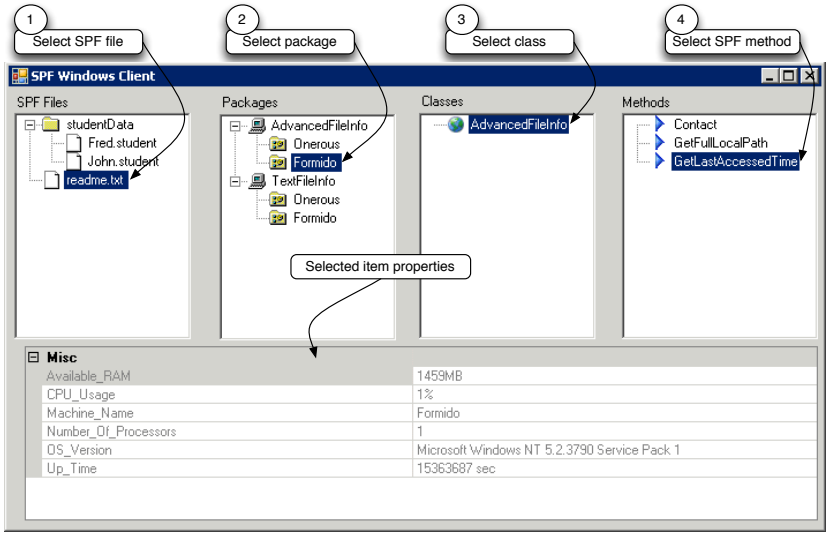

Figure 7.7: Shows the SPF Windows client GUI built upon the client API layer. The four steps show how the user selects a method which is associated with a data file. As objects are selected their properties are displayed in the lower region of the GUI.

- Step 3

  When a package is selected in the *packages* tree, the *classes* tree is populated. This is a list of all the classes contained within the selected package. In this example there is just one class, `AdvancedFileInfo`.

- Step 4

  When a class is selected in the *classes* tree, the *methods* tree is populated. When the `AdvancedFileInfo` class is selected the three methods shown in example 7.1 are displayed. The user can then select a method and execute it using the process described in section 7.6.2.

7.6.2 Executing an SPF method

Once a user has selected the SPF method to execute, it can then be invoked and the results returned. When a user selects a method in the *SPF Windows client* shown in figure 7.7 the *execute SPF method* window appears as shown in figure 7.8.

The four steps required to execute a method and obtain the results shown are described below:

- Step 1

  Since a class can have many constructors the user can select the constructor. This is an optional step as the SPF will select a constructor with a single input parameter over others as the SPF can execute this without user intervention. Where a constructor only takes one parameter, the file name is automatically passed to the constructor. If more than one parameter is required the SPF

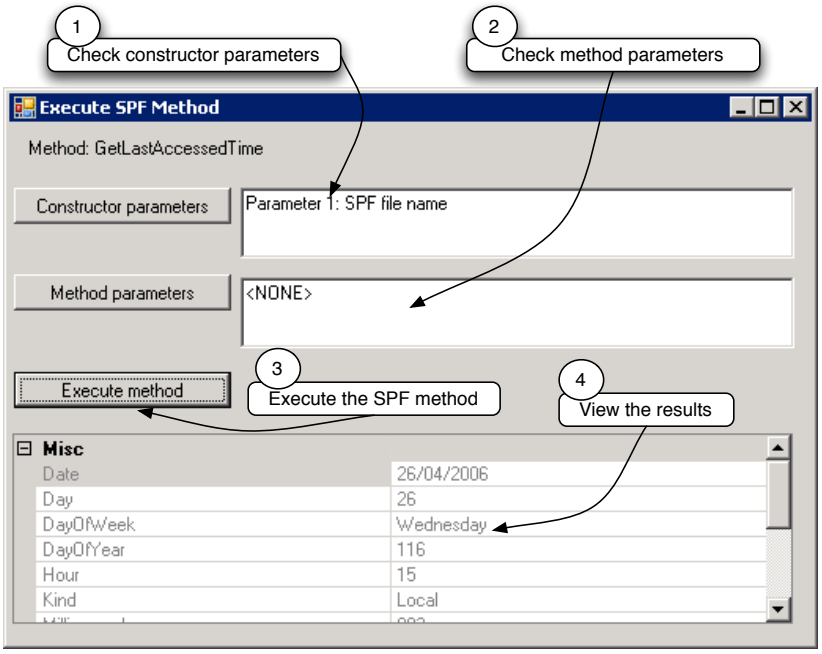

Figure 7.8: Shows the GUI to execute an SPF method. When a user selects a method this window provides the user with optional constructor and method parameter modifications. The results of a method execution are displayed in the lower properties window.

will look for a suitable candidate, failing that it will resort to passing the file name in as the first parameter. Subsequent parameters are the responsibility of the user. In the GUI interface, users can supply primitive parameters using the constructor parameters option. In this example the class only has one constructor which takes the file name, thus the default options are used.

- Step 2

  The default behaviour of the SPF is to execute a method without any parameters. However users can provide primitive parameters by selecting the method parameter option. In this example the method does not take any parameters.

- Step 3

  Once a user has set the optional constructor and method parameters it can be invoked using the execute method button. This will cause the storage service where the file is located to request the DLL Manager to create an instance of the class and invoke the selected method. The results from this method are then serialised and passed back to the client layer using web services. The client layer returns the results object to the calling interface, in this example the GUI.

- Step 4

  To demonstrate that the results of the method are returned as an object we display the properties of the object in the results view as shown in figure 7.8. It is expected that a user will take the return object and utilise it in a calling application. The GUI is intended to demonstrate the capabilities of the underlying API.

7.7 File object method features

In this section we look at all the FOM features and capabilities in turn and describe how the SPF supports each. Where applicable we use the example code described in section 7.5 and refer to the GUI discussed in section 7.6.

7.7.1 User data submission

All the storage services run on the host machine's file system and the structure of the files remain unchanged. Users are free to add and remove files from the storage service's root directory.

Simulation and experimental data can be written directly into the storage service using the user's preferred method. The API supports importing data files, although users do not have any control over where the files are stored.

7.7.2 User code submission

User codes must be compiled into a class library (DLL) which is common practice in the .NET framework.

The DLLs are stored in the `DLL_HOME` directory located in each storage service. Users can either manually copy the DLL into this directory or use the API to import the DLL. This only makes the SPF aware of the DLL but does not associate it with any files or file types. In section 7.7.3 we show how users can associate DLLs with data files.

The user's code can be written in any of the supported .NET languages resulting in the SPF's ability to support language independence.

7.7.3 Code association

Users have the ability to associate code with data files using the client API as shown in figure 7.9.

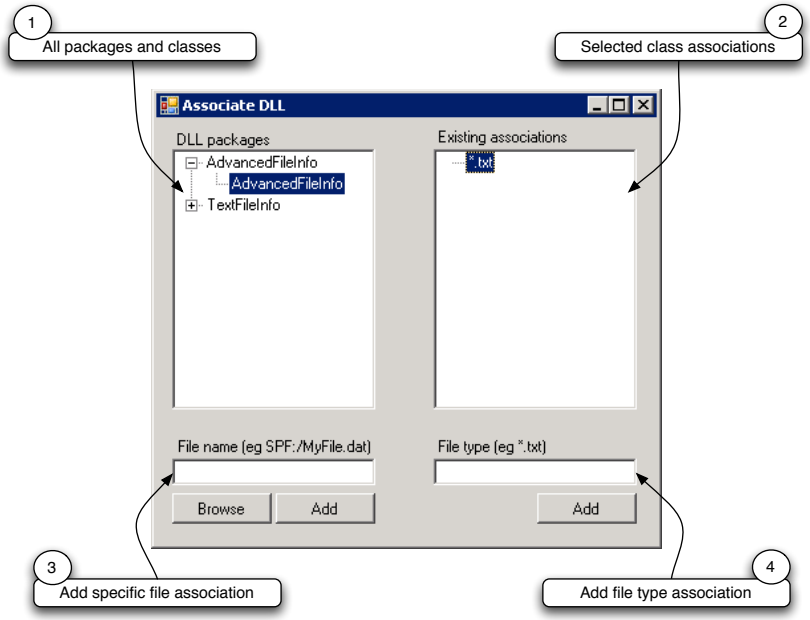

Figure 7.9: Shows the GUI used to associate users' code to data files and file types.

- Step 1

  This tree view displays all the packages and classes stored in the SPF. Code association is linked to the class and not the package. In this example the `AdvancedFileInfo` package has one class of the same name.

- Step 2

  When a class is selected in step 1, the associated files and file types are displayed. All the files and types shown in this window will automatically become associated with the `AdvancedFileInfo` class; in this example all the files with a `txt` extension.

- Step 3

  To associate additional files with the currently selected class, users can add full SPF file names, *e.g.* a user can associate the file `SPF:/MyFile.dat` with this class and it will appear in the window shown in step 2. Users must enter a full SPF file name and path, which can be obtained using the browse button and by selecting the appropriate file. This method is used to associate user's code with data files on a one-to-one basis.

- Step 4

  Users can associate the selected class with any file of a particular type. In this example entering `*.txt` will associate the `AdvancedFileInfo` class with all text files.

Directories and files are treated the same in the SPF, thus users can associate code on a one-to-one basis with directories.

7.7.4 Dynamic method discovery

The example shown in section 7.5 demonstrates how users can discover methods using the client API. All the methods associated with a data file can be listed, queried and executed. Although the GUI shown in figure 7.7 provides a visual insight into the SPF, all the features are programmable using a .NET language.

The FOM shell execution feature has been eliminated as the SPF permits the returning of data from all .NET languages as objects.

Results as objects or files

When an SPF method is executed the results are returned as .NET objects as shown in figure 7.8. These objects allow the user to programmatically integrate the SPF into existing applications.

Users cannot directly return SPF method results as file objects. If the SPF method creates a file when executed this will then appear in the same directory as the data file. This mechanism can be used to retrieve data from SPF methods as file objects.

7.7.5 Load balancing

The load balancing is managed in the API layer. When an SPF method is called the API looks at all the storage services and selects the one with the lowest CPU load. The rules which determine how a machine is selected are stored in a single class. Further SPF optimisations can include additional rules to manage the load balancing. For example, rules which take the available RAM, network speed and available storage can be added to the load balancing rules.

7.7.6 Security

All the SPF user accounts are created and managed by the Windows operating system. The SPF security is managed at the operating system level ensuring that the data and hosting machines remain robust against malicious users. Relying on the host Operating System (OS) ensures that users do not have to learn any new mechanisms and relies on tried and tested code. The security features built into most OS are thoroughly tested and normally updated as new threats arise.

The security is only as good as the configuration of the host OS, but these are well discussed issues. Users are often familiar with the pitfalls of configuring their OS of choice or can acquire support from existing knowledge bases.

Data

When data is deposited it is marked as read-only for all users and read-write for the depositing user. Other users can be given permission to alter data files using the OS user permission settings. Users can change the file permissions to suit their task using the methods currently used.

Code execution

All the code executed in the SPF framework runs as a restricted user. The SPF, by default only supports managed code (see section 6.4.4). This reduces the users ability to execute malicious code on any of the SPF storage services.

Users can see all the methods available any data file, the data file integrity is managed using the OS file permissions. The users code is restricted by the OS user account under which it is executed and the managed code only option reduced the opportunity to compromise security, for example by using buffer overflow hacking attempts.

Transport security

Users have the ability to turn on transport security features in the SPF. For example the MTOM data encryption shown in section 6.4.2 can be used to encrypt data sent to the storage services. The .NET framework supports Simple Object Access Protocol (SOAP) extensions to encrypt web service data.

In addition the option to run reliable message transacted messages is available using the web service standards shown in section 6.4.2.

These features can be enabled or disabled, as appropriate to the application.

7.7.7 Method results cache

When an SPF method is executed the results are requested from a storage service. Each storage service is completely autonomous and is responsible for invoking the DLL Manager. The storage service infrastructure is shown in figure 7.2. This provides a good opportunity to cache previously computed results.

Every time an SPF method is executed the calling parameters and the results are stored by the DLL manager. If the method is executed with the same parameters the cached results are returned.

The cache is only valid for a single storage service to ensure that data is kept consistent. If the data in a file is changed all the cached method results are removed.

7.7.8 Other features

The remaining FOM features described in chapters 3 and 6 are discussed below.

- Code quality assurance

  The SPF provides the same code quality assurance as the FODB prototype shown in chapter 4. Users submit managed classes which are made available to all users. A user can only change their own data or submitted code, but the SPF relies on peer review and user accountability to avoid malicious code.

- Cascading methods

  As with the FODB prototype shown in chapter 4 the SPF supports cascading methods. Any SPF methods common to all files in a directory appear on the directory object. This method will invoke all the methods on all files in the directory, returning the result as a filename-result tuple.

- Data distribution

  The SPF supports data distributed across many machines and is demonstrated in section 7.5.

- Remote method execution

  The location of any data is transparent to the SPF. This results in complete support for any remote execution of queries.

7.8 Discussion

The SPF demonstrates all the FOM features and provides an in-depth look at their capabilities.

When users submit code it is copied to a single storage service. Multiple submissions are required if the user wishes to make the code available on many machines. The client API supports methods to copy user code to other machines. Currently this copies the user's compiled library to a different machine. It is possible to automatically copy all users' code to all storage services but this does not deal with code dependencies. It is for this reason that code exists only on the machine where it is deposited. Determining what applications are available on each SPF service and only replicating user code to machines with the appropriate applications is beyond the scope of this research.

The SPF results cache is only flushed when a data file has been altered. Since an SPF method may return large volumes of data the cache can quickly grow. Currently the cache is limited by available storage space. It would be beneficial to change this cache so that only methods which are time consuming are cached.

There needs to be a mechanism to limit the size of the cache and rules dictating which cached results are removed first.

The web service reliable transaction standards shown in section 6.4.2 are suitable for future SPF implementations. Currently the SPF layer manages the web service messages, but the next logical progression is to move this logic from the application layer to the transport protocol layer. This would reduce the need to monitor, retry and handle messages. As the SPF gets larger these features would be beneficial.

7.9 Summary

In this chapter we demonstrate all the FOM features using a fully-functional prototype called the SPF. This prototype is implemented using the .NET framework and integrates data across a distributed environment.

We demonstrate the SPF capabilities using two user examples written in different programming languages. The SPF provides the ability to execute methods on remote machines and returns the results as .NET objects. The capabilities of the SPF underlying client API are demonstrated with the use of a GUI.

Using this interface we show how users can locate data and view its associated methods. The GUI can invoke SPF methods and display the results.

In chapter 8 the SPF prototype is compared with previous FOM implementations.

Chapter 8

Evaluation

8.1 Introduction

In chapter 3 we propose the File Object Method (FOM) as a mechanism for managing data. This is implemented in the File Object Database (FODB) prototype described in chapter 4 where we demonstrate the FOM capabilities.

In chapter 5 we utilise a limited set of FOM features to manage computational chemistry simulation data in the BioSimGrid project.

Using the FODB and BioSimGrid implementations of the FOM we then propose alterations to the FOM as described in chapter 6. These alterations are used to produce a final implementation-independent specification of the FOM, which is implemented in chapter 7 in a project called the Storage and Processing Framework (SPF).

The FODB, BioSimGrid and the SPF implementations of the FOM all support different features. The FODB implements all the basic FOM features, the BioSimGrid project supports a limited but distributed set of FOM features, whilst the SPF implements all the FOM features.

Each implementation is architected differently, utilising different technologies, programming languages and FOM features.

In this chapter we look at each of these implementations and compare their supported features. We show how the FOM supports the objectives of this work outlined in chapter 1 and discuss the limitations of the FOM.

8.2 Overview

The FOM is designed as a method to assist users with data management. It proposes the concept of associating programming methods with data files. This concept allows users to browse data files and view any appropriate methods that are capable of processing the data. Users of the FOM can utilise code that they may previously have been unaware. It also ensures that the methods shown are appli-

cable to the selected data file. The FOM is a concept which outlines the features to which any implementation must adhere. All the FOM features are shown in table 8.1.

The first FOM implementation shown in chapter 4, called the FODB , implements a sub set of these features. The FODB is intended as a proof of concept and aims to demonstrate the FOM features using sample data.

The BioSimGrid project is establishing a worldwide repository for simulation results as well as providing a data analysis framework. BioSimGrid manages multiple Tebibyte (TiB) of data and is implemented using a reduces version of the FOM. The FOM features implemented demonstrate the capability of the FOM to manage large volumes of data in a distributed environment.

The FODB demonstrates the FOM capabilities and the BioSimGrid project demonstrates the importance of distributed data. Using the observations from these projects, we proposed modifications to the FOM. These modifications include the ability for the FOM to operate across a distributed environment. This includes remote method execution, data security and a method results cache as shown in section 6.3. These modifications complete the FOM concept and produce a generic data management framework capable of managing large volumes of data , securely across a distributed environment.

In chapter 7 we provide a full FOM implementation called the SPF. This differs from the previous two implementations as it implements all the FOM features, including the FOM modifications described in chapter 6. The SPF is implemented using the Microsoft .NET framework and is capable of supporting user code in many different languages.

The capabilities of these three FOM implementations are described below.

8.3 Implementation feature comparison

The three FOM implementations implement a different subset of FOM features. The FODB demonstrates the key FOM features such as method discovery and the ability for users to submit their own code into the framework. The main limitations are to do with security, data distribution and support for multiple languages.

The BioSimGrid project provides a reduced FOM implementation capable of managing a large volume of production data. The features are limited to provide assurances about the data security and integrity. It provides an example of the FOM in a production project utilised by scientists across the UK and USA. Users are not able to submit their own code and there is no capability for caching method execution results.

The SPF framework is the first implementation to fully support multiple language user code submission. Using the .NET framework we can provide assur-

Table 8.1: Shows which of the features described in section 3.4 are implemented in each of the three implementations of the FOM. Full support for a feature is indicated with a ✓and partial support is shown as ○. Features left blank are not supported.

FOM feature	FODB	BioSimGrid	SPF
User data submission	✓	○	✓
Dynamic method discovery	✓		✓
Shell method execution	✓	✓	✓
Cascading methods	✓		✓
Code quality assurance		✓	
Data distribution		✓	✓
Remote method execution			✓
Load balancing			✓
Method results cache			✓
User code submission			
— Language dependence	✓	○	
— Language independence			✓
Method execution			
— Results as files	✓	○	○
— Results as objects	✓	✓	✓
Code association			
— File method association	✓	○	✓
— File type method association	✓		✓
— Directory method association	✓		✓
Security			
— Data	○	✓	✓
— Code execution		○	✓
— Transport security		○	✓

ances about the security of the data, code and transmission of data. The SPF also implements an example load balancing capability and a method results cache. These features provide a secure, efficient mechanism for users to access data across a distributed environment.

All the FOM features described in chapters 3 and 6 are listed in table 8.1. This table provides an overview of the features supported by each FOM implementation.

8.4 Objectives

The objectives identified in chapter 1 provide a basis for the FOM. The objectives are designed to ensure that any work utilises the appropriate technology and is

appropriate for the target community. In this section we look at each objective and evaluate the FOM implementations.

8.4.1 Data management framework

Large volumes of data need to be managed. In this objective we aim to produce a framework capable of managing large volumes of data.

The FODB does not manage data across distributed resources and is limited to a single instance. The implemented framework is capable of managing users code and is the basis for future implementations.

The BioSimGrid implementation currently manages multiple TiB of data across resources scattered across the United Kingdom and United States of America. Although this is a limited FOM implementation it provides a working example of a large volume data management framework.

The SPF is based on the BioSimGrid and FODB implementations. The SPF manages files using the native operating system file system to provide a secure scalable data storage framework.

8.4.2 Resource utilisation management

One of the key FOM objectives is to optimise the management of existing resources to promote the reuse of computation results. All data stored in the FOM can be made available to other users reducing recalculation.

The SPF supports the caching of method results which can speed the user's calculations. If an SPF method is invoked with the same parameters the repeat calculation is eliminated by using results from the cache.

8.4.3 Data organisation

The FOM organises data by associating related code to data files. The data can be retrieved and manipulated using the framework. This linking of code to data assists the user by ensuring that the code is suitable for the data.

8.4.4 Locate data

As data volumes increase, they become fragmented across many resources, locations and architectures. Locating data becomes more complex and time consuming.

In this work we aim to enhance the user's ability to locate data. This has been achieved in the SPF implementation of the FOM.

Users can view the structure of all the files stored in the SPF transparently of their location. The FOM is designed to ensure that users can keep the original directory structure, file names and data format.

The FOM improves the user's ability to locate and manage data but has two limitations:

- Users are unable to search files for content, *i.e* locate all text files in the FOM containing a specific phrase.
- Users have to manage the directory structure of data files. Users are free to deposit data into the FOM by directly accessing the FOM storage devices. Thus two users can create files with the same name. The SPF will flag the data as out of sync and prompt the user. Using a fully qualified directory structure may be useful, *e.g.* `SPF:\\<company name>\<project>\<data>`

There is a need to specify which directory structure users should create, but due to the distributed nature of the SPF implementation control is difficult. This can be overcome using a different underlying distributed file system.

8.4.5 Promote collaborations

The FOM aims to promote collaborations between users. This is achieved by the exposure of all data within the FOM to all users. The FOM ensures that the users code is correctly executed on the associated data, negating the need for users to understand, compile or install the users code. The FOM enables the sharing of data and code seamlessly across a distributed environment.

The FOM differs from a distributed file system as users can process the data using associated code. This is important as data in general is useless without an understanding of the format or knowledge of applications to extract the information from the data. In the FOM, users are automatically provided with code that is capable of processing the data files. The code is already configured so the user can easily execute and obtain results, *i.e.* the SPF provides a single click execution GUI.

8.4.6 Generic data

The FOM is data domain independent. Only the BioSimGrid implementation of the FOM is domain specific. This is due to the nature of the project and the vast quantity of data management required.

The FODB and SPF implementations of the FOM are capable of storing data regardless of their domain. Any data that can be stored in a file or series of files can be stored. The FOM specifies that the data is kept in the original format and structure to ensure that any implementations are capable of dealing with any user data.

8.4.7 Generic resources

The FOM definition is discussed independently from any implementation to ensure that the proposed method is not fixed to any operating system or architecture. In the three FOM implementations we show how the FOM can operate using a variety of scenarios.

The FODB can operate on any machine supporting Python, *i.e.* Linux, Windows and Mac OS X. This implementation is a standalone version which makes cross platform support easier.

The BioSimGrid implementation is written in Python and supported on Linux (SuSE 9.X, Open-SuSE 10.0 (*Novell*, 2006)). Portability is limited as the BioSimGrid application requires many platform specific applications.

The SPF implementation is the most restrictive in terms of portability, but provides most support for FOM features. The restrictions originate from the .NET framework which has limited cross-platform support. In section 6.4.5 we described some of the ongoing work to provide better platform support. As the SPF is implemented using web services it is possible to extend the client and storage manager services to other platforms using a different programming language *e.g.* Java. Only the storage service currently requires the full .NET framework support, in order to support the user's code. Section 8.5.4 discusses the portability limitations.

8.4.8 Utilise open standards

Wherever possible we have adopted open standards to maximise adoption of the FOM. Below we list the key areas where open standards have been included

The FOM takes advantage of open standards wherever possible to maximise the uptake and extensibility of the concept.

For example, below we list the key areas where open standards have been utilised.

- Web services

  The SPF is built using a Service-Oriented Architecture (SOA) resulting in a Web Service for each package. Web Services use a series of open standards to establish communication and transfer data. For example Web Services Description Language (WSDL), Extensible Markup Language (XML), and Simple Object Access Protocol (SOAP). This web service architecture ensures that alternative FOM implementations are capable of interfacing with the existing architecture.

- MTOM

  Message Transmission Optimisation Mechanism (MTOM) complies with open standards and it utilised to transfer data between web services, as discussed

in section 6.4.2

- .NET Framework

  The SPF is implemented using the .NET Framework which is itself an open standard. Although many alternative implementations are not currently available we can expect to see alternatives in the future, as discussed in section 6.4.

8.5 Limitations and improvements

In this work we proposed the FOM as a data management concept and provided implementations to demonstrate its features. In this section we discuss these implementations and their limitations.

8.5.1 Versioning

When a user submits data or code it is stored and managed by the FOM. Users are then free to make changes, providing they have the appropriate permissions. This can lead to versioning issues. If code is being used by another user and the owner updates the functionality, or if data changes during analysis, the results may be inaccurate or confusing.

There is a need for the FOM to provide code and data versioning to ensure that calculations are repeatable. Versioning data can consume large amounts of storage and is not appropriate for volatile data. Tools like shadow copy, discussed in section 2.7 may be appropriate for data versioning.

The FOM can be modified to version any user's code by retaining previously deposited versions instead of replacing them. Providing code versions complicates the FOM and provides users with more options. The FOM aims to be simple and unintrusive, so versioning needs to provide users with appropriate default behaviour and the versioning needs to be provided as an advanced option.

8.5.2 Results caching

The FOM cache implemented in the SPF provides users with an inbuilt automatic optimisation capable of returning pre-computed results instantly. The current FOM cache, implemented in the SPF has several limitations.

The cache is local to each storage service, the user only benefits if a service with a cached result is queried. If another service is used then the result is recomputed. The storage layer is de-centralised to assist with reliability and load balancing. Storage services are selected based on a set of rules which require some performance tuning. For example, if an SPF method takes one day to compute a result and the result is already cached, the next invocation should result in the cached copy being returned.

Currently a storage service with a high load is not selected even if there is a cached copy, thus the result is recomputed. Storage nodes are not required to have high availability but are compensated for with the use of data replications. It is not clear what should happen when a storage service is scheduled for down time. Should the removal of a storage service result in the migration of cached results, particularly compute intensive results.

All method calls are cached. The cache size differs between storage services but when it is full, the oldest results are deleted. This is not always the desired behaviour, there is a need to calculate the optimal cached results. For example should compute intensive results have priority over frequently retrieved results? Caching all method calls is also not optimal. For example, a call to retrieve the raw data will result in the data being copied to the cache. This consumes space and is not compute intensive. The cache needs to take into account: i) the result size, ii) time to compute and iii) frequency of cache hits.

The SPF provides the ability to change the load balancing and caching options. In future work, it is beneficial to have an automatic caching formula as it is not appropriate for users to manage the cache.

8.5.3 Code and data security

Securing data, code and analysis in the FOM is a big issue. Although the FOM aims to exchange data and code, users require assurances about validity of any data. Executing user's code can be dangerous and pose a risk to FOM data.

Data is secure only if it is stored on a secure machine. The distributed FOM implementation makes security difficult. If the data is processed on a machine, then at some point the data will exist in an unencrypted state in memory. Therefore, if the physical security of a machine is compromised, the data is compromised. This makes securing a distributed network inherently impossible. Although there are discussions about hardware encryption support in processors which may assist with this issue (*Becker et al.*, 2003).

8.5.4 Portability

The FOM aims to encourage collaborations and exchange of data between users. Users often have an operating system and platform preference. If the FOM restricted users to only particular platforms or operating systems this would alienate many users. The aim is to bring advanced database features to the user without changing the users preferred working environment. The FOM has to support most operating systems, most platforms and the most common programming languages.

The final FOM implementation, the SPF is the most advanced but provides limited portability. This is based on the .NET framework, which has limited support

for non-Microsoft operating systems. In section 6.4 we describe the ongoing work to support .NET framework portability.

8.5.5 Volatility of variables

The FOM does not deal with volatile variables or functions. For example if a user were to submit code to return the local time of a machine. The FOM method results cache would cache the results, rendering the results obsolete.

To overcome this we propose the option of disabling the results cache, or restricting the caching of results to non-volatile results.

In `C#` as with many other programming languages, variables can be marked as volatile by using the `volatile` keyword (*Horstmann*, 2005). The cache can then avoid the caching of these results, however it relies on the programmer to markup variables appropriately.

8.6 Summary

In this chapter we look at the different FOM limitations and compare the capabilities of each. The FODB is a proof of concept which demonstrates the FOM features. The BioSimGrid project demonstrates the FOMs capability to manage large volumes of data. The SPF provides a final FOM implementation capable of supporting all the FOM features, each implementation builds upon findings of the previous.

We demonstrate which of the FOM features are supported across the three implementations. These are displayed in table 8.1 and show how the designs have evolved.

Finally we discuss how the final proposed FOM meets the objectives described in chapter 1.

Chapter 9

Summary

In this work we identify current data management challenges and investigate how non-technical scientific users can benefit from existing data management technologies. Often existing data remains unpublished, stored inaccessibly and in proprietary data formats. This results in the loss and repetition of many calculations which is an inefficient use of computational resources.

In this work we outlined a series of objectives to produce a data management framework for scientific users, with the ability to leverage existing technologies. These objectives show the requirements of a suitable framework concept. The previous chapter contained a detailed evaluation of the framework we have developed. Most notably the framework concept has to be platform and programming language independent and require minimal effort from the users. We propose that users retain computationally expensive data and encourage collaborations to reuse data where possible.

Processing power and storage technologies are advancing at different rates. In the future a single unit of processing power will be responsible for managing a larger quantity of data. The existing scientific workflow needs to change to ensure that resources are utilised optimally. We discuss the suitability of databases and file systems for managing data and demonstrate where each technology is suitable. For example small frequently accessed metadata is best suited to a database whilst large binary data is best suited to a file system.

We propose the concept of a File Object Method (FOM) framework which associates users code with data files. This ensures that users can manipulate and process any data file without the need for complex data format specifications. The FOM provides a framework which is transparent to users and keeps existing data and workflows unchanged. At any point in the FOM, users are able to access and manipulate the data files. The FOM is implementation-independent and outlines the features any implementation must support.

The FOM provides users with the ability to seamlessly share data manipulation code. The FOM can discover code associated with any data file and subsequently

execute methods, returning data objects to the user. This ensures that appropriate code is invoked on data files, regardless of their format. Users can add their existing code into the FOM for the benefit of other users and browse the FOM for code associated with any particular data file.

The first FOM implementation, called the File Object Database (FODB) provides a working prototype which successfully demonstrates the FOM capabilities. The FODB is implemented in Python, with which many scientific users are familiar. We demonstrate how users can submit data and execute FODB methods. Users can interface with the FODB using the commandline or programmatically from within a Python script. The FODB restricts users to Python and operates on a stand-alone machine.

Having demonstrated the feasibility of the FOM concept we introduce a limited set of FOM features into the BioSimGrid project. The BioSimGrid project is establishing a globally accessible molecular dynamics simulation repository. The project currently manages multiple Tebibyte (TiB) of data and supports users internationally. We utilise the FOM capabilities to manage the underlying data files which are distributed across nine geographical locations.

BioSimGrid provides a valuable insight into the capabilities of the FOM and demonstrates its capabilities for managing large volumes of data.

Using the experiences gained from the FODB prototype and the BioSimGrid implementation we propose changes to the FOM. For example, we propose that the FOM must operate across distributed resources, support multiple programming languages and is capable of caching results from previously executed FOM methods. Distributing data in the FOM enhances data accessibility and provides the ability to support data redundancy.

We have identified the need to move computation closer to the data to avoid expensive data transfers (both in terms of time and expense), the FOM provides this transparently to the end user.

Using the FOM proposed changes and the existing FOM implementations we introduce a final FOM implementation called the Storage and Processing Framework (SPF). The SPF implements all the FOM features including results caching and remote method execution across distributed resources. The SPF is implemented using the .NET Framework which provides the SPF with the capability to manage user code in over twenty different programming languages.

We demonstrate the FOM features using the SPF and an example scenario in both C# and Visual Basic. The SPF method discovery, execution and retrieval of results, from both local and remote resources is demonstrated using a graphical interface built upon the SPF.

The SPF remote method execution capability utilises the .NET Framework remoting features which limits the SPF to serialisable objects. Using the .NET Framework enhances the SPF by permitting the execution of SPF methods from any .NET

supported language. This ensures that the data objects are compatible with the users language of choice. The SPF transparently supports remote data access and returns data objects supported by any .NET Language.

The FOM produces a generic data management framework which is described conceptually and implemented in various prototypes.

9.1 Further work

The current proposed FOM demonstrates an ideal framework to address the user's data management issues. The areas of interest for future work are highlighted below.

- Currently user's code is stored on a machine selected by the user. It would be beneficial to copy the code to additional FOM storage services. Since the code is small it is easy to replicate and improves the FOM load balancing. Currently the FOM does not support user code prerequisites. For example, user's code requiring an installed package, library or tool cannot easily be replicated to other storage services. There is a need to store any operating system, hardware and other software requirements along with the user's code. This ensures that if any code is replicated, it is guaranteed to run on another appropriately selected machine.

- Data replication is expensive and requires careful scheduling to ensure that many FOM storage services do not attempt to replicate at the same time. An alternative to scheduling is using a Quality of Service (QoS) to prioritise traffic. As replications can take time, the users are exposed to a potential data loss risk. The availability of nodes and time taken to replicate needs to be considered when storing data with a high availability demand.

- The SPF implementation of the FOM is based on a basic distributed file system. Distributed file systems are well researched and many well tested products exist. Future FOM implementations need to support existing distributed file systems.

- The directory structure is managed by users creating and removing directories. As the volume of users increases it becomes increasingly more difficult to control the directory structure. This may be overcome with the use of an appropriate distributed file system.

- Many scientists are familiar with particular operating systems, for example Linux, Mac OS X and many UNIX variants. The SPF is implemented using the .NET framework which is mainly supported on the Windows platform. For the FOM the become successful it will require support for other operating

systems. The various .NET framework implementations require investigation and integration into future FOM implementations.

9.2 Summary

Scientists and engineers increasingly require the ability to archive, retrieve and manage the ever growing data volumes generated by complex computational simulations.

By adopting data management frameworks, such as the one we have outlined which, in this thesis provides seamless and transparent data management services in the environments in which they perform their simulations. They will be better able to tackle the next generation of complex problems.

Furthermore such technologies are the key to unlock greater scientific insight from data.

Appendix A

Computing resources

A.1 BioSimGrid hardware

Each of the six BioSimGrid sites have identical production machines and Oxford and Southampton have an additional testing machine.

Table A.1, shows the hardware used in each of the six BioSimGrid sites. Each BioSimGrid site has storage attached, table A.2 shows the configuration and storage capacity of each.

CPU	2 X AMD Athlon MP 2400 [2.00GHZ]
Motherboard	MSI K7D Master Dual AMD MP
RAM	2 Gibibyte (GiB)
Hard disk	Maxtor 250 Gibibyte (GiB) IDE

Table A.1: BioSimGrid hardware.

Hard drives	250GB Maxtor SATA
Number of HDD	16
Configuration	RAID 5 + 1 Hot spare
Usable storage	3.6TiB

Table A.2: RAID configuration and specifications

Appendix B

BioSimGrid

B.1 Flat files

The `FlatFileManager` class manages the two versions of the flat file accessing methods. The code utilised within the BioSimGrid project to manage all the data files is listed below.

Example B.1: Shows the Python code used to manage the flatfiles, this version shows how the data is spread across multiple files and how each file is managed.

```
'''
BioSimGrid    http://biosimgrid.org/    team@biosimgrid.org
Copyright 2003, 2004, 2005 University of Oxford and
University of Southampton. If you use this software in any way, please cite:
Kaihsu Tai, Stuart Murdock, Bing Wu, Muan Hong Ng, Steven Johnston, Hans
Fangohr, Simon J. Cox, Paul Jeffreys, Jonathan W. Essex, Mark S. P.
Sansom (2004) BioSimGrid: towards a worldwide repository for
biomolecular simulations. Org. Biomol. Chem. 2:3219-3221
http://dx.doi.org/10.1039/b411352g
http://eprints.ouls.ox.ac.uk/archive/00000804/

This program is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 2 of the License, or (at your
option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA
http://www.fsf.org/licensing/licenses/gpl.txt
'''
"""
Author:
Steven Johnston <s.j.johnston@soton.ac.uk>
Descriptions:
This is the point of entry for flatFile data retrieval.
This class is here to manage the different Traj databases, as well and
```

```
manage the connections, it will return the flatDatabase object for a
given location.
$Header: /var/deposit/codebase/BioSim/Database/FlatFileManager.py,v
    1.12 2005/06/23 13:38:35 sjj Exp $
"""
from BioSim.Database.flatDatabase import FlatDatabase
from BioSim.Database import SRBFlatDatabase
import Numeric,logging , shutil,os, time
from BioSim.Settings import ServerSettings
from BioSim.Database.Importer import DbaseImporter
class FlatFileManager:

    def __init__(self, userSettings):
        self.CVS_REVISION='$Revision: 1.12 $'
        self.revision = self.CVS_REVISION.split(" ")[1]
        self.log = logging.getLogger('general')
        self.log.info( "New flat file manager" + " Version : " + str(
            self.revision))
        self._userSettings = userSettings #input suer settings object
        self.serverSettings = ServerSettings.ServerSettings()
        self.gidPrefix = self.serverSettings.getGIDPrefix() # you will
             need to change your settings file for your site, not this
            !
        self.userBasePath = self._userSettings.getOutputPath()
        self.basePath = self.serverSettings.getDataFilesRootDir()
        self.tempPath = self.serverSettings.getDataFilesTempDir()
        #connection pool, has a max of about 200 files for all trajs
        self.__connPool = {}

    def getConn(self, gid):
        """
        Get flat database connection
        """
        try:
            return self.__connPool[gid]
        except KeyError:
            sfDB =  SRBFlatDatabase.SRBFlatDatabase(gid ,userSettings=
                self._userSettings,serverSettings =self.serverSettings
                )
            self.log.debug("Hit connection cache..." + str(gid))
            self.__connPool[gid] = sfDB
            return sfDB

    def __calcname(self,gid):
        return self.gidPrefix+str(gid)

    def getNewDatabase(self, gid):

        """
        Set New bioSim.Database connection
        """
        return SRBFlatDatabase.SRBFlatDatabase(gid , userSettings=self
            ._userSettings ,serverSettings =self.serverSettings,
            create=True)

if __name__=="__main__":
    #Commandline usage, example
    from BioSim.Database import FlatFileManager
    from BioSim.Logger import BioSimLog
    from BioSim.Settings import UserSettings
    BioSimLog.BioSimLog()
    usr = UserSettings.UserSettings("sjj")
    ffM = FlatFileManager.FlatFileManager(usr)
```

Example B.2: Shows the Python code to access the frames of data stored using the Python pickle method.

```
'''
BioSimGrid    http://biosimgrid.org/    team@biosimgrid.org
Copyright 2003, 2004, 2005 University of Oxford and University of
    Southampton
If you use this software in any way, please cite:
Kaihsu Tai, Stuart Murdock, Bing Wu, Muan Hong Ng, Steven Johnston,
    Hans
Fangohr, Simon J. Cox, Paul Jeffreys, Jonathan W. Essex, Mark S. P.
Sansom (2004) BioSimGrid: towards a worldwide repository for
biomolecular simulations. Org. Biomol. Chem. 2:3219-3221
http://dx.doi.org/10.1039/b411352g
http://eprints.ouls.ox.ac.uk/archive/00000804/

This program is free software; you can redistribute it and/or modify
    it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 2 of the License, or (at your
option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
    along
with this program; if not, write to the Free Software Foundation, Inc
    .,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA
http://www.fsf.org/licensing/licenses/gpl.txt


$Header: /var/deposit/codebase/BioSim/Database/flatDatabase.py,v 1.5
    2005/06/09 13:24:20 muanhong Exp $

'''

import  os, random, sys, tempfile, time,array
import Numeric as Num
from struct import unpack
import pickle
import logging

class FlatDatabase:

    def __init__(self, fileName, create=False ,  userSettings=None):
        """Sets the name of the database, if the database exists then
            the metadata
        is loaded, else the database is created"""
        self.CVS_REVISION='$Revision: 1.5 $'
        self.revision = self.CVS_REVISION.split(" ")[1]
        self._userSettings = userSettings
        self.log = logging.getLogger('general')
        self.log.debug("New flatDatabase created Filename: "+ str(
            fileName)+ " Version : " + str(self.revision))
        self.create = create
        self.metaDataExt = ".dat"
        self.binExt =  ".bin"
        self.path , self.shortFileName = os.path.split(fileName)
        self.log.debug("Path: " + self.path)
        self.fileName  = fileName
        self.__open( fileName )

```

```
56      def __open( self, fileName ):
57          """Internal method! called by init() this method sets the max
                file size and
58          initialises the internal data structures, checks to see if the
                 database exists
59          and calls the appropriate methods"""
60          #store name of database
61          self.databaseName = fileName
62                  #Set max file size!
63                  self.fileSizeLimit = 500000000 #bytes :-)
64                  #create blank globals
65                  self.metaData = {}
66                  self.dataFileList= {}
67                  #See if database exists or needs creating
68                  if(os.path.isfile(str(self.databaseName)+self.
                        metaDataExt)):#idx is key file
69              self.__openExisting()
70                  else:
71              if (self.create):
72                  self.log.debug("Creating new database: " + str(self.
                        fileName))
73                  self.__createNew()
74              else:
75                  self.log.critical("Could not find database " +  str(
                        self.fileName) + "  Did you put the full path?, if
                         you are trying to create the database you need to
                         set the create flag in the constructor to TRUE")
76                  raise str(self.revision) + "Invalid database Name : "
                        + str(fileName)
77
78
79      def __createNew(self):#private
80          self.log.debug( "Creating new bioSim.Database...")
81          self.numberOfAtomsPerFrame = -1
82          self.emptyDatabase = True
83
84      def __openExisting(self):#private!
85                  self.log.debug("Database exists, opening...")
86                  self.emptyDatabase = False
87                  #load metadata
88                  self.__loadMetaData()
89
90      def __loadMetaData(self):
91          """
92          This looks at the .DAT file and loads the hash table with the
                metadata into memory (internal methos)
93          """
94                  metaDataFile = open(self.databaseName+self.metaDataExt
                        , 'rb' )
95                  self.metaData = pickle.load(metaDataFile)
96                  self.frameSize = self.metaData["frameSize"]
97                  self.framesPerFile = self.metaData["framesPerFile"]
98                  self.fileSizeLimit = self.metaData["fileSizeLimit"]
99                  self.numberOfAtomsPerFrame = self.metaData["
                        numberOfAtomsPerFrame"]
100                 self.typeCode =self.metaData["typeCode"]
101                 metaDataFile.close()
102                 self.log.debug( "Metadata loaded:"+ str(self.metaData)
                        )
103
104     def __saveMetaData(self):
105         """
106         This saves the metadata as the database is closing
107         """
108                 metaDataFile = open(self.databaseName+self.metaDataExt
                         , 'wb' )  #possible BUG, yes you were right
```

```
109                 self.metaData["frameSize"] = self.frameSize #saves
                        time if in here and i use a global
110                 self.metaData["framesPerFile"] = self.framesPerFile
111                 self.metaData["fileSizeLimit"] = self.fileSizeLimit
112                 self.metaData["numberOfAtomsPerFrame"] = self.
                        numberOfAtomsPerFrame
113                 self.metaData["typeCode"] = self.typeCode
114                 pickle.dump(self.metaData,metaDataFile)
115                 metaDataFile.close()
116                 self.log.info("MetaData saved" + str(self.metaData))
117
118     def __saveAll(self):
119         """
120      Insures that all open databases are CLOSED , used in
            deconstructor
121      """
122      self.log.debug("List of file keys" + str(self.dataFileList))
123
124         for keys in self.dataFileList.keys():
125         self.log.debug("closing file:  " + self.databaseName+ str(keys
                )+self.binExt)
126         self.dataFileList[keys].close()
127
128
129     def __del__(self):
130         """
131         Deconstructor, saves metadata and closes all open files
132         """
133         self.log.debug("----Deconstructor----")
134         self.__saveAll()
135         try:
136             if self.create:
137                 self.__saveMetaData()
138         except Exception , desc:
139             self.log.critical("Could not save the metadata for " +
                    self.fileName +  " this could be because , no file
                    space, you did not add an y frames to a new database ,
                     therefore there is not metadata...")
140                 #close all data files
141
142
143     def __calculateFrameMetaData(self,data):#private
144         """
145         Takes the first frame supplied as an example for all the rest,
                calculates metadata from this.
146         """
147         self.log.debug("Empty database, taking this frame as example
                for all rest")
148         tempFileName = tempfile.mktemp()
149         fil = open(tempFileName,'wb')
150         try:
151                 Num.dump( data, fil)
152                 self.frameSize = fil.tell()
153                 self.log.debug("Each frame is bytes long" + str(self.
                        frameSize))
154         finally:
155                 fil.close()
156                 os.remove(tempFileName)
157                 self.emptyDatabase = False
158                 x,y =  Num.shape(data)
159                 self.numberOfAtomsPerFrame = x* y
160                 self.framesPerFile = self.fileSizeLimit / self.
                        frameSize
161                 self.typeCode = data.typecode()
162                 self.log.info("Number of frames per file :"  + str(
                        self.framesPerFile) +"Number of atoms per frame:"+
```

```
                    str(self.numberOfAtomsPerFrame))
163         if(self.framesPerFile == 0):
164                 self.log.critical( "MaxFileSize has to be at least the
                        size of a single frame!")
165                 exit
166         self. __saveMetaData()
167
168
169     def __getFileForFrame(self,frameID,readOnly):#private
170         """
171         This returns the file for containing the requested frame, it
                also moves the pointer to that location in the file.
172         """
173                 fileNumber = frameID /  self.framesPerFile
174                 #see if file exists
175          if ~( fileNumber in self.dataFileList.keys()):
176             currFilename = self.databaseName+str(fileNumber)+".bin"
177
178             if(os.path.isfile(currFilename)):
179                 if(readOnly):
180                     self.dataFileList[fileNumber] =  open(currFilename
                            , 'rb' ) #sjj - +
181                 else:
182                     self.dataFileList[fileNumber] =  open(currFilename
                            , 'r+b' ) #sjj - +
183             else:
184                 if(readOnly):
185                     msg = "Cant find file and not able to create one (
                            create flag not set)"
186                     self.log.debug(msg)
187                     raise msg
188                 else:
189                     self.dataFileList[fileNumber] =  open(currFilename
                            , 'w+b' )
190
191                 #get position in the file where it should be/go
192                 positionInFile = (frameID-(self.framesPerFile *
                        fileNumber))*self.frameSize
193                 #print "Position:",positionInFile
194                 #seek to location
195                 self.dataFileList[fileNumber].seek(positionInFile)
196             self.log.debug("List of data files " + str(self.
                    dataFileList))
197                 return self.dataFileList[fileNumber]
198
199     def __setitem__( self, frameID, data ):
200         self.setFrame(frameID,data)
201
202     def setFrame(self,frameID, data):
203         self.log.debug("Setting frame: " + str(frameID))
204         if (self.emptyDatabase):
205                 self.__calculateFrameMetaData(data)
206         if (self.numberOfAtomsPerFrame != -1):
207                 x , y = Num.shape(data)
208                 if ( x * y != self.numberOfAtomsPerFrame):
209               msg = "Frame needs to be the same size, ignoring ADD"
210               self.log.error(msg)
211               msg = "expected : " + str(self.numberOfAtomsPerFrame) +
                      " got : " + str(Num.shape(data))
212               self.log.error(msg)
213               raise msg
214                 if(data.typecode() != self.typeCode):
215              msg = "This data is not of the same type expected %c got
                     %c, ignoring data" %(self.typeCode , data.typecode())
216              self.log.error(msg)
217              raise msg
```

```
218
219             file = self.__getFileForFrame(frameID,False)
220             Num.dump( data, file )
221             file.flush()
222             self.log.debug("Dump, at no" + str(frameID))
223             #self.log.debug("Location " + str(file.tell()))
224             self.log.debug("CTR: " + str( frameID )+ "Getframe test" + str
                    (Num.shape(self.getFrame(frameID))))
225
226
227         def getFrame(self,frameID):
228             file = self.__getFileForFrame(frameID,True)
229             if(file == None):
230                 self.log.debug("Invalid ID range, or file not there")
231
232                 return None
233             try:
234                 data = Num.load(file)
235             except EOFError, description:
236                 self.log.debug("End of file , hence frame is invalid")
237                 return None
238             return data
239
240
241         def __getitem__( self, frameID ):
242              return self.getFrame(frameID)
243
244
245         def getAtoms(self,frameID, listOfAtomID): # ~100 frames /sec
246             currFrame = self.getFrame(frameID)
247              output = Num.ones((len(listOfAtomID),3), self.typeCode)
248              for ctr in range(len(listOfAtomID)):
249                  output[ctr] = currFrame[listOfAtomID[ctr]]
250              return output
251
252         def getAtom(self,frameID,aid):
253             return self.getAtoms(frameID,[aid])
254
255         def getFullMetFileName(self):
256             name = self.fileName
257             self.log.debug("Full meta file name  : " + name)
258             return name
259
260         def getListOfFiles(self):
261             fileList = os.listdir(self.path)
262             returnList = []
263             for fileName in fileList:
264                 currFileName = self.path + os.sep + fileName
265                 if os.path.isfile(currFileName):
266                     if currFileName.find(self.shortFileName):
267                          self.log.debug("File name" + fileName)
268                          returnList.append(currFileName)
269             self.log.debug("Full file List " +str( returnList))
270             return returnList
271
272         def getShortName(self):
273             ### this is the base traj name...
274             self.log.debug("Short filename for a traj : " + self.
                    shortFileName)
275             return self.shortFileName
```

Example B.3: Shows the Python code to access serialised frames of data from files stored in SRB.

```
'''

BioSimGrid    http://biosimgrid.org/    team@biosimgrid.org

Copyright 2003, 2004, 2005 University of Oxford and University of
    Southampton

If you use this software in any way, please cite:
Kaihsu Tai, Stuart Murdock, Bing Wu, Muan Hong Ng, Steven Johnston,
    Hans
Fangohr, Simon J. Cox, Paul Jeffreys, Jonathan W. Essex, Mark S. P.
Sansom (2004) BioSimGrid: towards a worldwide repository for
biomolecular simulations. Org. Biomol. Chem. 2:3219-3221
http://dx.doi.org/10.1039/b411352g
http://eprints.ouls.ox.ac.uk/archive/00000804/

This program is free software; you can redistribute it and/or modify
    it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 2 of the License, or (at your
option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
    along
with this program; if not, write to the Free Software Foundation, Inc
    .,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA
http://www.fsf.org/licensing/licenses/gpl.txt

'''
"""Testing the storage of the database date in 'raw' binary files to
reduce access time and storage requirements.


See log at end of file for details.
"""

import Numeric , time, StringIO
from BioSim.Base import BsgBase
from SRBInterface import SRBFile, SRBCon
from BioSim.Database.BinaryFiles import FrameSerialise

class SRBFlatDatabaseFile(BsgBase.BsgBase):

    def __init__(self,srbConnection,  fileName, metData , create=False
        , collection='', userSettings=None):
        """Sets the name of the database, if the database exists then
            the metadata
        is loaded, else the database is created"""
        self.CVS_REVISION='$Revision: 1.15 $' .split(" ")[1]
        #init the base class
        BsgBase.BsgBase.__init__(self)
        self._userSettings = userSettings
        self._collection = collection
        self._create = create
        self._srbConnection = srbConnection
        self._metadata = metData
        self._converter =  FrameSerialise.FrameSerialise(self.
            _metadata[self._metadata.ARRAY_WIDTH_STR],self._metadata[
```

```
            self._metadata.DATA_TYPE_STR])
57          self.log.debug("SRB flatDatabase.. ")
58          self.log.debug("Data type in met: " + str( self._metadata[self
                ._metadata.DATA_TYPE_STR]))
59          self._fileName = fileName
60          self._srbFile = self.__openFile(self._fileName)
61
62      def __openFile(self, fileName):
63          self.log.debug("Opening file : " + str(fileName))
64          srbFile  = SRBFile.SRBFile(self._srbConnection,fileName,
                create=self._create,collection=self._collection)
65          if srbFile.created:
66              self._metadata[self._metadata.FILE_NAMES_STR].append(self.
                    _fileName)
67              self._metadata.saveMetadata()
68          return srbFile
69
70      def __setitem__(self,frameID, data):
71          self.setFrame(frameID, data)
72
73      def setFrame(self,frameID, data):
74          binData = self._converter.numericArrayToData(data)
75          offset = frameID * self._metadata[self._metadata.
                FRAME_SIZE_STR]
76          self._srbFile.write(binData, offset)
77
78      def __getitem__(self,frameID):
79          return self.getFrame(frameID)
80
81      def getFrame(self,frameID):
82          frSize = self._metadata[self._metadata.FRAME_SIZE_STR]
83          offset = frameID *  frSize
84          dat = self._srbFile.read(frSize, offset)
85          arr = self._converter.dataToArray(dat , self._metadata[self.
                _metadata.NUM_ATOMS_STR] )
86          return arr
87
88      def getAtoms(self,frameID, atomList):
89          frSize = self._metadata[self._metadata.FRAME_SIZE_STR]
90          offset = frameID *  frSize
91          pairs = Numeric.reshape(atomList,(len(atomList)/2,2))
92          numberSize = self._metadata.DATA_TYPE_SIZES_DICT[self.
                _metadata[self._metadata.DATA_TYPE_STR]]
93          data = StringIO.StringIO()
94          numAtoms = 0
95          for pair in pairs:
96              posInFrame = (pair[0] * numberSize*3)
97              atomOffset = posInFrame +offset
98              numAtoms += pair[1]
99              dataLen = pair[1] * numberSize * 3
100             if (posInFrame + dataLen) > frSize:
101                 msg = "Hou have asked for an atom that is not in the
                        frame, Max frame size :" + str(frSize)
102                 msg += " You asked for: " + str(posInFrame + dataLen)
                        + " For frameInFileID : " + str(frameID)
103                 msg += " Using atom list : " + str(atomList)
104                 self.log.critical(msg)
105                 raise msg
106             dat = self._srbFile.read(dataLen, atomOffset)
107             data.write(dat)
108         arr = self._converter.dataToArray(data.getvalue() , numAtoms )
109         self.log.debug( arr.typecode())
110         return arr
```

B.2 BioSimGrid web portal

The BioSimGrid web portal provides users with an online interface to the BioSimGrid analysis tools. Users can select the tools, trajectories and frames to be analysed. Figures B.1 —B.5 show the web portal capabilities from logging in, to generating a BioSimGrid analysis script.

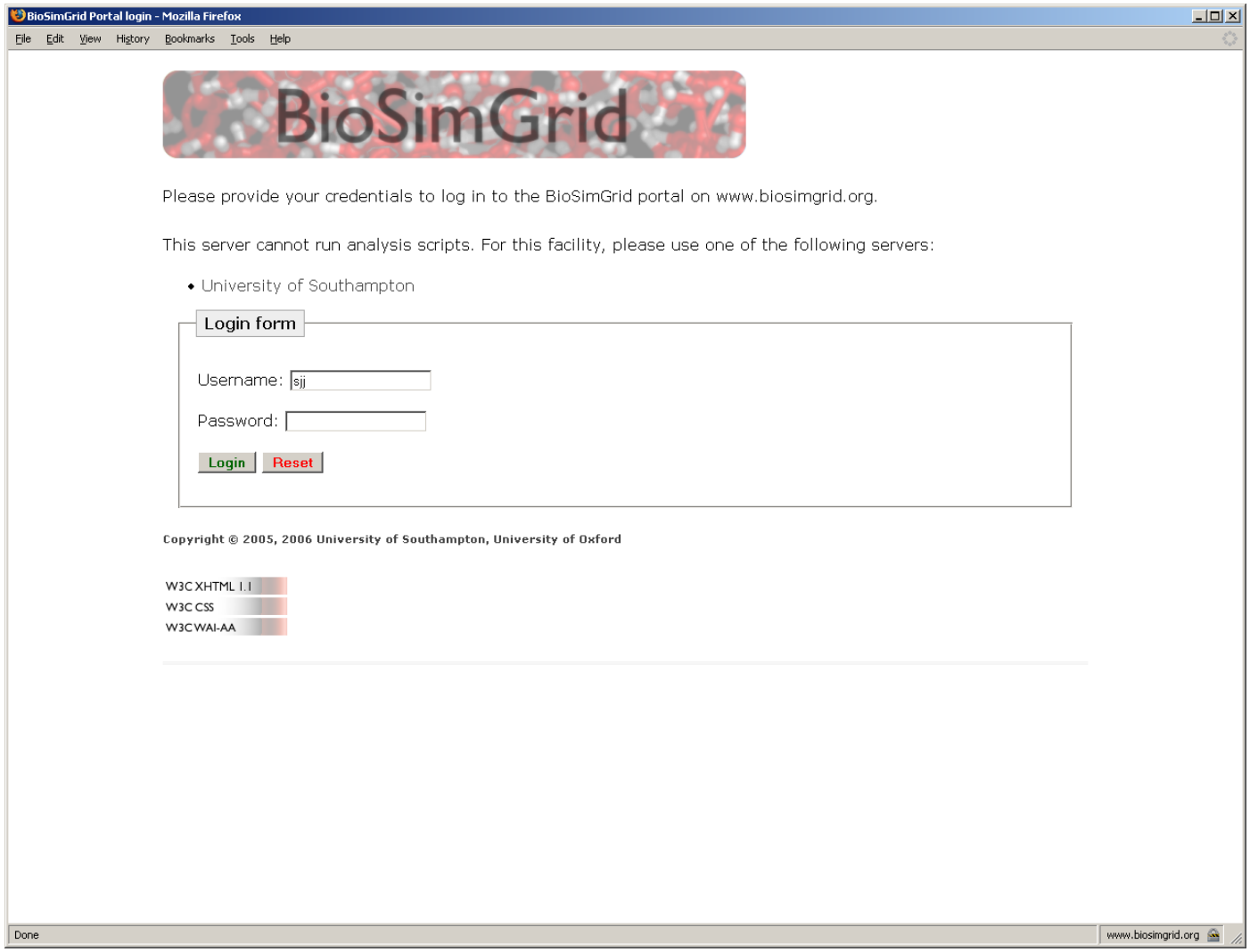

Figure B.1: BioSimGrid web portal login page

Figure B.2: Selecting a trajectory using the BioSimGrid web portal.

Figure B.3: Each analysis tool can have many methods. The web portal displays the tool and its description.

Figure B.4: Parts of a trajectory can be selected using the *frame collection* web page.

Figure B.5: The web portal generates a Python script which is capable of being run on any of the BioSimGrid nodes. Users are able to alter this script before it is executed.

Bibliography

Adachi Y. Database schema design strategy in relational database. *SECOM Co., Ltd. Intelligent Systems Lab* (2001).

Alapati SR. *Expert Oracle 9i Database Administration*. Apress (2003).

Allen DW, Coles SJ, Light ME and Hursthouse MB. Synthesis and X-ray crystal structures of organotri(2- furyl)phosphonium salts: effects of 2-furyl substituents at phosphorus on intramolecular nitrogen to phosphorus hypervalent coordinative interactions. *Inorganica Chimica Acta*, **357**, 1558–1564 (2004).

Alur N, Almaraz J, Meira PD and Yorita R. *WebSphere Information Integrator Content Edition: Planning, Configuration, and Monitoring Guide*. IBM Redbook (2005).

Anderson R. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons Inc. (2001).

Apple. Technical note TN1150 : HFS Plus volume format. *Developer Connection* (2004).

Apple. Mac OS X v.10.4 Tiger overview: New features, APIs, and frameworks. *Developer Connection* (2005a).

Apple. Spotlight overview. *Developer Connection* (2005b).

Apple (2006). [Online; accessed 1-June-2006], `www.apple.com/macosx`.

Arinaminpathy Y, Beckstein O, Biggin P, Bond P, Domene C, Pang A and Sansom M. Large scale biomolecular simulations: Current status and future prospects. *Proceedings of UK e-Science All Hands Meeting* (2003).

Armstrong E, Ball J, Bodoff S, Carson DB, Evans I, Green D, Haase K and Jendrock E. *The J2EE 1.4 Tutorial*. Sun Microsystems (2005).

Atkinson B, Della-Libera G, Hada S, Hondo M, Hallam-Baker P, Kaler C, Klein J, LaMacchia B, Leach P, Manferdelli J, Maruyama H, Nadalin A, Nagaratnam N, Prafullchandra H, Shewchuk J and Simon D. Web services security. *IBM Developerworks* (2002).

Bachman C. Integrated data store. *DPMA Quarterly* (1965).

Baeza-Yates R and Ribeiro-Neto B. *Modern Information Retrieval*. Addison-Wesley, 1^{st} edition (1999).

Barry D. Transparent persistence (2004). [Online; accessed 1-August-2004], `www.service-architecture.com`.

Barry DK. *Web Services and Service-oriented Architecture: The Savvy Manager's Guide*. Morgan Kaufmann (2003).

Becker E, Buhse W, Günnewig D and Rump N. *Digital Rights Management: Technological, Economic, Legal and Political Aspects*, volume 2770. Springer (2003).

Beckett D. The design and implementation of the Redland RDF application framework. *Tenth International World Wide Web Conference* (2001).

Benson DA, Karsch-Mizrachi I, Lipman DJ, Ostell J and Wheeler DL. Genbank: update. *Nucleic Acids Research*, **32** (2004).

Berendsen H, van der Spoel D and van Drunen R. GROMACS: A message-passing parallel molecular dynamics implementation. *Comp. Phys. Comm. 91* (1995).

Berman H, Westbrook J, Feng Z, Gilliland G, Bhat T, Weissig H, Shindyalov I and Bourne P. The protein data bank. *Nucleic Acids Research*, **28** (2000).

Bilorusets R, Box D, Cabrera LF, Davis D, Ferguson D, Ferris C, Freund T, Hondo MA, Ibbotson J and Jin L. Web services reliable messaging protocol. *OASIS Standards* (2005). [Online; accessed 17-June-2006], `schemas.xmlsoap.org/ws/2005/02/rm/`.

Black G, Schuchardt K, Gracio D and Palmer B. The extensible computational chemistry environment: A problem solving environment for high performance theoretical chemistry. 17^{th} *Annual ACM International Conference on Supercomputing* (2003).

Boardman RP, Johnston S, Essex JW, Ng M, Fangohr H, Tai K and Sansom MSP. BioSimGrid on the desktop (2006). In preparation for submission.

Bond PJ and Sansom MSP. The simulation approach to bacterial outer membrane proteins. *Molecular Membrane Biology*, **21**(3), 151–161 (2004).

Borret R. XML database products (2004). [Online; accessed 1-August-2004], `www.rpburret.com/xml`.

Britton D, Cass A, Clarke P, Coles J, Doyle A, Geddes N, Gordon J, Jones R, Kelsey D, Lloyd S, Middleton R, Pearce S and Tovey D. GridPP: Meeting the particle physics computing challenge. *UK e-Science All Hands Conference, Nottingham* (2005).

Brooks BR, Bruccoleri RE, Olafson BD, States DJ, Swaminathan S and Karplus M. CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, **4**, 187–217 (1983).

Brown E. An overview of SQL Server 2005 Beta 2 for the database administrator. *Journal* (2004).

Bush I and Sunderland A. Scalable eigensolvers on hpcx: Case studies. *HPC Research Technical Report HPCxTR0510* (2005).

Butler GF, Baird WP, Lee RC, Tull CE, Welcome ML and Whitney CL. The Global Unified Parallel File System (GUPFS) project: FY 2003 activities and results. *Lawrence Berkeley National Laboratory*, **16**, 49–61 (2004). Paper LBNL52456_2003.

Cabrera LF, Copeland G, Feingold M, Freund RW and Freund T. Web services coordination. *IBM Developerworks* (2005a).

Cabrera LF, Copeland G, Feingold M, Freund RW and Freund T. WS-AtomicTransaction specification . *IBM Developerworks* (2005b).

Cabrera LF, Copeland G, Feingold M, Freund RW and Freund T. WS-BusinessActivity specification . *IBM Developerworks* (2005c).

CERN. Hierarchical databases (2002). [Online; accessed 21-December-2005], `www.db.web.cern.ch`.

CERN. The world's largest particle physics laboratory (2005). [Online; accessed 1-November-2005], `www.cern.ch`.

Chamberlin DD and Boyce RF. SEQUEL: A structured english query language. *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264 (1974).

Chan P, Lee R and Kramer D. *The Java Class Library*. Addison-Wesley Professional, 2^{nd} edition (1998).

Chappell D. Introducing Indigo: An early look. *Chappell and Associates* (2005).

Chinnici R, Hadley M and Mordani R. *The Java API for XML-Based Web Services (JAX-WS) 2.0*. Sun Microsystems, Inc., 2^{nd} edition (2006).

Clark J and DeRose S. XML Path language(XPath) version 1.0 (1999).

Codd E. *A Relational Model of Data for Large Shared Data Banks*, volume 13, pages 377–387. Communications of the ACM (1970).

Cox S, Chen L, Campobasso S, Duta M, Eres M, Giles M, Goble C, Jiao Z, Keane A, Pound G, Roberts A, Shadbolt N, Tao F, Wason J and Xu F. Grid Enabled Optimisation and Design Search (GEODISE). *e-Science All Hands, Sheffield* (2002).

Cox S, Fairman M, Xue G, Wason J and Keane A. The Grid: Computational and data resource sharing in engineering optimisation and design search. *International Conference on Parallel Processing Workshops*, page 207 (2001).

Custer H. *Inside the Windows NT File System*. Microsoft Press, 1^{st} edition (1994).

Cwalina K. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison Wesley (2005).

Darrow B. MySQL reference manual (2004). [Online; accessed 1-August-2004], `www.mysql.com`.

Date C. *An Introduction to Database Systems*, chapter Further Normalisation I:1NF, 2NF, 3NF, BCNF, pages 348–379. Addison-Wesley, Reading , Massachusetts, 7^{th} edition (2000a).

Date C. *An Introduction to Database Systems*. Addison-Wesley, Reading , Massachusetts, 7^{th} edition (2000b).

Date C. *An Introduction to Database Systems*, pages 150–198. Addison-Wesley, Reading , Massachusetts, 7^{th} edition (2000c).

Date C. *An Introduction to Database Systems*, chapter Further Normalisation II: Higher Normal Forms, pages 389–416. Addison-Wesley, Reading , Massachusetts, 7^{th} edition (2000d).

Deitel HM, Deitel PJ and Liperi JP. *Python How to Program*. O'Reilly & Associates, book and CD-ROM edition (2002).

DotGNU (2006). [Online; accessed 12-June-2006], `www.dotgnu.org`.

Drenth J. *Principles of Protein X-ray Crystallography*. Springer, 2^{nd} edition (2002).

DuBois P. Using XML with MySQL, document revision: 1.01 (2003). [Online; accessed 1-August-2003], `www.kitebird.com/articles/mysql-xml.html`.

Edwards J, McCurley K and Tomlin J. An adaptive model for optimizing performance of an incremental web crawler. *IBM Almaden Research Center* (2002).

Eisenberg A, Melton J, Kulkarni K, Michels JE and Zemke F. SQL:2003 has been published. *SIGMOD Rec.*, **33**(1), 119–126 (2004).

England K. *Microsoft SQL Server 2000 Performance Optimization and Tuning Handbook*, pages 163–173. Digital Press, 1^{st} edition (2001).

Essex DJ. Towards the dynome: Adding a 4^{th} dimension to the protein database by terascale simulation. *EPSRC Grant Reference: EP/D000173/1* (2005).

Feuerstein S and Pribyl B. *Oracle PL/SQL Programming*. O'Reilly Media, Inc. (2005).

Foley M. WinFS to be available on Windows XP. *Microsoft Watch* (2005). [Online; accessed 7-February-2006], `www.microsoft-watch.com`.

Foster I. Globus toolkit version 4: Software for service-oriented systems. *IFIP International Conference on Network and Parallel Computing*, pages 2–13 (2005).

Foster I and Kesselman C. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications* (1997).

Foster I and Kesselman C, editors. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 2. Morgan-Kaufman (1999a).

Foster I and Kesselman C. The Grid: Blueprint for a new computing infrastructure. *Morgan Kaufmann* (1999b).

Foster I, Kesselman C, Nick J and Tuecke S. The Physiology of the Grid: An open Grid services architecture for distributed systems integration. *Open Grid Service Infrastructure WG, Global Grid Forum* (2002a).

Foster I, Kesselman C and Tuecke S. The anatomy of the Grid: Enabling scalable-virtual organisations. *International Journal on Supercomputer Applications* (2001).

Foster I, Kesselman C and Tuecke S. The anatomy of the Grid: Enabling scalable virtual organisations. *International J. Supercomputer Applications*, **15** (2002b).

Frawley W, Piatetsky-Shapiro G and Matheus C. Knowledge discovery in databases: An overview. *AI Magazine*, pages 213–228 (1992).

Freed N and Borenstein N. Multipurpose internet mail extensions. *The Internet Engineering Task Force, RFC-2045* (1996). [Online; accessed 20-February-2006], `www.ietf.org/rfc/rfc2045.txt`.

Freeman RG and Hart M. *Oracle9i RMAN Backup and Recovery*, chapter Differential vs. Incremental backups. McGraw-Hill Professional, 1^{st} edition (2002).

Frey J, Tannenbaum T, Livny M, Foster I and Tuecke S. Condor-G: A computation management agent for multi-institutional Grids. *In Proceedings of the* 10^{th} *IEEE Symposium on High Performance Distributed Computing (HPDC10)*, pages 55–63 (2001).

Gallardo D. *Java Oracle Database Development*. Prentice Hall (2002).

Getz K, Litwin P and Reddick G. *Microsoft Access 2 Developer's Handbook*. Sybex Inc, paperback edition (1994).

Ghemawat S, Gobioff H and Leung ST. The Google file system. *In Proceedings of the* 19^{th} *ACM Symposium on Operating Systems Principles* (2003).

Giampaolo D. *Practical File System Design with the Be File System*. Morgan Kaufmann Publishers, Inc (1999).

Gong L, Mueller M, Prafullchandra H and Schemers R. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. *USENIX Symposium on Internet Technologies and Systems* (1997).

Gordon A. *The .NET and COM Interoperability Handbook*. Prentice Hall (2002).

Gould L, Zanevsky A and Kline K. *Transact-SQL Programming*. O'Reilly Media, Inc. (1999).

Gowin J. Journalling file systems: An intro to Reiserfs (2000). [Online; accessed 1-August-2004], `pro.linuxorbit.com`.

GPL. GNU general public license (1991). [Online; accessed 7-December-2005], `www.gnu.org/copyleft/gpl.html`.

Gray J and Reuter A. *Transaction Processing : Concepts and Techniques*, chapter 12. Morgan Kaufmann, 1^{st} edition (1993).

Grochowski E. MR/GMR read head evolution. *San Jose Research center* (2003). [Online; accessed 21-December-2005], `www.hitachigst.com/hdd/hddpdf/tech/hdd_technology2003.pdf`.

Gulutzan P. Transaction logs (2003). [Online; accessed 1-August-2004], `www.dbazine.com`.

Haile JM. *Molecular Dynamics Simulation : Elementary Methods*. Wiley-Interscience, 1^{st} edition (1997).

Halfhill TR. A peek at OFS. *BYTE Digest* (1995).

Han J and Kamber M. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 1^{st} edition (2000).

Hand D, Mannila H and Smyth P. Principles of data mining. *MIT Press* (2001).

Harbottle J, Darcy M and Gillies J. IBM to help CERN build massive data Grid to understand origins of the universe. *IBM Grid Computing* (2003).

Harris S and Gibbins N. 3store: Efficient bulk RDF storage. *Workshop on Semantic Web Storage and Retrieval* (2003).

Henderson K. *The Guru's Guide to SQL Server Stored Procedures, XML, and HTML*, chapter 1, pages 4–7. Addison-Wesley Professional (2001).

Hertel C. *Implementing CIFS: The Common Internet File System*. Prentice Hall (2003).

Horstmann CS. *Core Java 2: Advanced Features*, chapter 1. Prentice Hall (2005).

Howard J, Kazar M, Menees S, Nichols D, Satyanarayanan M, Sidebotham R and West M. Scale and performance in a distributed file system. *ACM Trans. on Computer Systems*, pages 51 – 81 (1988).

Humphrey W, Dalke A and Schulten K. VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics*, **14**, 33–38 (1996).

IBM. IBM , website (2005). [Online; accessed 1-December-2006], `www.ibm.com`.

IEC. Letter symbols to be used in electrical technology - part 2: Telecommunications and electronics. *International Standard 60027-2* (2005).

IEEE. Ieee standard for binary floating-point arithmetic. *Institute of Electrical and Electronics Engineers, Inc* (1985). ANSI/IEEE Std 754-1985 (IEEE 754).

ISO. *Codes for the representation of names of countries and their subdivisions*. Niso Press, 1^{st} edition (1995). Part 2: Country subdivision code.

ISO. Information technology — database languages — SQL:Framework (SQL/Framework) (2003). ISO/IEC 9075-1:2003.

JACS. Journal of the american chemical society (2004). [Online; accessed 1-December-2005], `http://pubs.acs.org/journals/jacsat`.

Jagatheesan A and Moore R. Mass storage systems and technologies. 12^{th} *NASA Goddard/*21^{st} *IEEE Conference on Mass Storage Systems and Technologies* (2004).

Janmohamed Z, Liu C, Bradstock D, Chong RF, Gao M, McArthur F and Yip P. *DB2(R) SQL PL : Essential Guide for DB2*. IBM Press, 2^{nd} edition (2004).

Jiao Z, Wason J, Molinari M, Johnston S and Cox S. Integrating data management into engineering applications,. *Proceedings of UK e-Science All Hands Meeting, Nottingham, UK*, pages 687 – 694 (2003).

Johnson MK. Whitepaper: Red hat's new journalling file system: ext3 (2001). [Online; accessed 1-August-2004], `www.redhat.com/support/`.

Johnston S. Python-SRB interface (2005a). [Online; accessed 15-December-2005], `http://sourceforge.net/projects/pysrb`.

Johnston S. Storage resource broker (SRB), large scientific data and Python. *Europython 2005* (2005b).

Johnston S, Boardman R, Fangohr H and Cox S. Managing large volumes of distributed scientific data. *Journal of Grid Computing* (2006a). (Submitted).

Johnston S, Boardman RP, Ng M, Essex JW, Fangohr H, Tai K and Sansom MSP. BioSimGrid: Infrastructure, performance and applications (2006b). In preparation for submission.

Jones A. *C# Programmer's Cookbook*, chapter 1, pages 17–28. Microsoft Press (2003).

Jones R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons Ltd. (1996).

Jones S and Morris M. A methodology for service architectures. *OASIS SOA Adoption Blueprints* (2005).

Kandemir M, Li F, Chen G, Chen G and Ozturk O. Studying storage-recomputation tradeoffs in memory-constrained embedded processing. *Design, Automation and Test in Europe*, **2**, 1026–1031 (2005).

Kelso RJ, Buszczak M, nones ATQ, Castiblanco C, Mazzalupo S and Cooley L. Flytrap, a database documenting a gfp protein-trap insertion screen in Drosophila melanogaster. *Nucleic Acids Research*, **4** (2004).

Levesque H and Lakemeyer G. *The Logic of Knowledge Bases*, page 95. The MIT Press, 1^{st} edition (2001).

Levinson E. The MIME Multipart/Related Content-type. *Network Working Group* (1998).

Leyderman R. *Oracle C++ Call Interface*, chapter 5. Oracle Corporation, 2^{nd} edition (2002).

Litwin W. *Applications of Databases*, chapter Text retrieval database applications, page page 255. Springer (1994).

Litzkow M, Livny M and Mutka M. Condor - a hunter of idle workstations. In *8th International Conference of Distributed Computing Systems*, pages 104–111 (1988).

Lundh F. *Python Standard Library*. O'Reilly & Associates, book and CD-ROM edition edition (2001a).

Lundh F. *Python Standard Library*, chapter 1, page 8. O'Reilly Media, Inc, 1^{st} edition (2001b).

Mamone M. *Practical Mono*. APress (2005).

Martelli A, Ascher D and Ravenscroft A. *Python Cookbook*, chapter 18, pages 656 – 657. O'Reilly (2005).

Mattos NM, Darwen H, Cotton P, Pistor P, Kulkarni K, Dessloch S and Zeidenstein K. SQL99, SQL/MM, and SQLJ: An Overview of the SQL Standards. *IBM Database Common Technology* (1999).

McBride B. An introduction to RDF and the Jena RDF API (2005). [Online; accessed 17-June-2006], `jena.sourceforge.net`.

McKusick MK. *The Design and Implementation of the FreeBSD Operating System*. Addison Wesley (2004).

McLean S. *.NET Remoting*. Microsoft Press (2002).

McMillan G. Socket programming (2004). [Online; accessed 1-August-2004], `www.amk.ca/python/howto/`.

Menascé D. Composing web services:a QoS view. *IEEE internet comuting* (2004).

Menzies T and Hu Y. Data mining for very busy people. *IEEE Computer*, pages 18–25 (2003).

Michie D. Memo functions and machine learning. *Nature*, **218**, 19–22 (1968).

Microsoft. NTFS file system (2000). [Online; accessed 1-December-2005], `www.microsoft.com`.

Microsoft. *Microsoft SQL Server 2000 Resource Kit*, page 166. Microsoft Publishing (2001).

Microsoft. DIME: Sending Binary Data with Your SOAP Messages (2002a). [Online; accessed 1-August-2004], `msdn.microsoft.com/library/default.asp?url=/library/en-us/dnservice/html/service01152002.asp`.

Microsoft. Direct Internet Message Encapsulation (DIME) (2002b). [Online; accessed 1-August-2005], `msdn.microsoft.com/library/en-us/dnglobspec/html/draft-nielsen-dime-02.txt`.

Microsoft. Understanding DIME and WS-Attachments (2002c). [Online; accessed 1-December-2005], `msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dnarxml/html/dimewsattch.asp`.

Microsoft. Web Services Security (WS-Security) (2002d). [Online; accessed 1-August-2004], `www.verisign.com/wss/wss.pdf`.

Microsoft. WS-Attachments (2002e). [Online; accessed 1-January-2005], `msdn.microsoft.com/library/en-us/dnglobspec/html/draft-nielsen-dime-soap-01.txt`.

Microsoft. Introduction to shadow copies of shared folders. Technical report, Microsoft Corporation (2003).

Microsoft. Microsoft SQL server 2000 resource kit (2004). [Online; accessed 1-August-2004], `www.microsoft.com/sql/`.

Microsoft. Microsoft windows vista beta 1 fact sheet. *PressPass* (2005a). [Online; accessed 1-February-2006], `www.microsoft.com/windowsvista`.

Microsoft. Web Services Security – SOAP Messages with Attachments (SwA) Profile 1.1 (2005b). [Online; accessed 1-December-2005], www.oasis-open.org/committees/download.php/13288/wss-v1.1-spec-pr-SwAProfile-01.html.

Microsoft. .net framework conceptual overview. *.NET Framework Developer's Guide* (2006). [Online; accessed 7-February-2006], http://msdn2.microsoft.com/en-us/library/zw4w595w.aspx.

Mitchell JC and Apt K. *Concepts of Programming Language*, chapter 4. Addison Wesley (2005).

Moore G. Cramming more components onto integrated circuits. *Electronics*, **38**(8), 114–117 (1965).

Moore R, Marciano R, Wan M, Sherwin T and Frost R. Towards the interoperability of web, database, and mass storage technologies for petabyte archives. *Procs. Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies* (1996).

Mullins CS. *Triggers and DB2 Version 6*. DB2 Update — Xephon (1999).

Mullins CS. *DB2 Developer's Guide*. Sams, 5^{th} edition (2004).

Murdock S, Muan Hong Ng, Johnston S, Fangohr H and Essex J. Comparing the performance of a database, specific binary files and netCDF for data retrieval (2004). [Online; accessed 10-December-2005], www.biosimgrid.org.

Murdock S, Ng M, Johnston S, Fangohr H and Essex J. BioSimGrid: A distributed database for biomolecular simulations (2002). [Online; accessed 1-January-2005], www.biosimgrd.org.

Murdock SE, Tai K, Ng M, Johnston S, Wu B, Fangohr H, Laughton CA, Essex JW and Sansom MSP. Quality assurance for biomolecular simulations. *Journal of Chemical Theory and Computation* (2006). Sumbitted.

Nagar R. *Windows NT File System Internals : A Developer's Guide*. O'Reilly, 1^{st} edition (1997).

Newman H. Choosing the right file system for linux clusters (2003). [Online; accessed 5-December-2004], hpc.devchannel.org/.

Ng M, Johnston S, Murdock S, Wu B, Tai K, Fangohr H, Cox S, Essex JW, Sansom M and Jeffreys P. Efficient data storage and analysis for generic biomolecular simulation data. *In Proceedings of UK e-Science All Hands Meeting 2004*, pages 443–450 (2004).

Ng M, Johnston S, Wu B, Murdock S, Tai K, Fangohr H, Cox SJ, Essex JW, Sansom MSP and Jeffreys P. BioSimGrid: Grid-enabled biomolecular simulation data storage and analysis. *Future Generation Computer Systems*, **22**, 657–664 (2006).

Nickell S and Fergeau C. *GnomeVFS — File system Abstraction library*. The Free Software Foundation (2004).

Novell (2006). [Online; accessed 1-June-2006], openSUSE.org.

Object Management Group. *Common Object Request Broker Architecture: Core Specification*. Object Management Group (2004).

Oliphant TE. Guide to numpy. *Trelgol Publishing* (2006).

Oracle. PL/SQL user's guide and reference (2001). Release 9.0.1, Part Number A89856-01.

Oracle. Oracle9i Advanced Replication (2002). Release 2, Part No. A96567-01.

Oracle. Oracle Internet File System installation guide release 9.0.1.1.0 for Microsoft Windows NT/2000. *Oracle Internet File System Archive Documentation* (2003). Part Number A85272-03.

Oracle. Oracle database 10g release 2 documentation library (2005). [Online; accessed 1-December-2005],`www.oracle.com/technology/documentation`.

Oracle. Oracle Database Backup and Recovery Basics (2005a). 10g Release 2 (10.2), Doc No. B14192-02.

Oracle. *Oracle Database Backup and Recovery Basics*, chapter 1.7. Oracle, 2^{nd} edition (2005b). Part number B14192-03.

Otey M. *Microsoft SQL Server 2005 New Features*. McGraw-Hill Osborne Media, 1^{st} edition edition (2004).

Parhami B. *Computer Arithmetic: Algorithms and Hardware Designs*, chapter 17, pages 282–285. Oxford University Press (1999).

Patterson DA, Gibson GA and Katz RH. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Conference*, pages 109–116 (1988).

PCCP. Physical chemistry chemical physics (2004). [Online; accessed 1-December-2005], `www.rsc.org/`.

Peterson R. *Database Development With Jdbc, Odbc and SQL/SQLJ*. SAMS, 1^{st} edition (2001).

Phillips JC, Braun R, Wang W, Gumbart J, Tajkhorshid E, Villa E, Chipot C, Skeel RD, Kale L and Schulten K. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, **26**, 1781–1802 (2005).

Plante RL, Crutcher RM and McGrath RE. The NCSA astronomy digital image library: from data archiving to data publishing. *Future Generation Computer Systems*, **16**, 49–61 (1999).

Powers L and Snell M. *Visual Basic Programmer's Guide to the .NET Framework Class Library*, chapter Reading and Writing to Files and Streams. Sams publishing, 1^{st} edition (2002).

PSF. Python Software Foundation home page (2005).

Rajasekar A, Wan M, Moore R, Jagatheesan A and Kremenek G. Real experiences with Data Grids — Case studies in using the SRB. *Proceedings of 6th International Conference/Exhibition on High Performance Computing Conference in Asia Pacific Region (HPC-Asia)* (2002).

Ramakrishnan R, Gehrke J, Ramakrishnan R and Gehrke J. *Database Management Systems*, page 6. McGraw-Hill Science, 3^{rd} edition (2002).

Reiser H. Future vision whitepaper (1984). Revised 1993, [Online; accessed 12-December-2004], `www.namesys.com/whitepaper.html`.

Reiser H. Reiser4 is released (2004). [Online; accessed 1-August-2004], `www.namesys.com/`.

Rew R, Davis G, Emmerson S and Davies H. NetCDF users guide for C (1997).

Richter J. *Applied Microsoft .NET Framework Programming*. Microsoft Press (2002).

Rivest R. The MD5 message-digest algorithm. *RFC 1321, MIT LCS & RSA Data Security Inc* (1992).

Rizzo T. WinFS 101: Introducing the new Windows file system. *Microsoft Corporation* (2004).

Rosen K, Host D, Farber J and Rosinski R. *Unix: The Complete Reference*, chapter 7. McGraw-Hill Osborne Media, 1^{st} edition (1999).

Russinovich M. NT internals: Inside Win2K NTFS. *Windows2000 Magazine* (2000). [Online; accessed 1-August-2005], `www.msdn.microsoft.com`.

Sammes AJ and Jenkinson B. *Forensic Computing: A Practitioner's Guide*. Springer-Verlag London Ltd (2007).

Sansom MS. E-science & biomolecular simulations:BioSimGrid & IntBioSim. *BBSRC Grant holders workshop* (2005).

Sansom PM. IntBioSim: An integrated approach to multi-level biomolecular simulations. *BBSRC Grant Reference: BBSB16011* (2004).

Schneier B. *Applied Cryptography*, chapter 18, pages 436–441. J. Wiley and Sons (1996).

SDC. Development and management of a computer-centered data base (1963). System Development Corporation.

Sears R, Ingen C and Gray J. To BLOB or not to BLOB: Large object storage in a database or a filesystem. *Microsoft Research* (2006).

SETI@home. Seti at home (2006). [Online; accessed 23-February-2006], `http://setiathome.ssl.berkeley.edu`.

Shepler S, Callaghan B, Robinson D, Thurlow R, Beame C, Eisler M and Noveck D. *Network File System (NFS) version 4 Protocol*. The Internet Engineering Task Force (2003). RFC3530.

Silverstein RM, Webster FX and Kiemle D. *Spectrometric Identification of Organic Compounds*. John Wiley & Sons, 7^{th} edition (2005).

Spealman J, Hudson K, Craft M and Corporation M. *Planning, Implementing, and Maintaining a Microsoft Windows Server 2003 Active Directory Infrastructure*. Microsoft Press (2003).

Stephenson P. *Oracle Database Lite 10g Technical White Paper*. Oracle Corporation, Redwood Shores, CA 94065, U.S.A., 1^{st} edition (2005).

Stix G. The triumph of the light. *Scientific American*, **284**(1), 69–73 (2001).

Stone J, Gullingsrud J, Grayson P and Schulten K. A system for interactive molecular dynamics simulation. In JF Hughes and CH Séquin, editors, *2001 ACM Symposium on Interactive 3D Graphics*, pages 191–194. ACM SIGGRAPH, New York (2001).

Stonebraker M. *Readings in Database Systems*, chapter Access Methods: B-Trees, R-Trees & GiSTs. Morgan Kaufmann Publishers, 2^{nd} edition (1993).

Sun Microsystems. *Java 2 Platform Standard*. Sun Microsystems, 5^{th} edition (2004).

Szalay AS, Gray J, Thakar AR, Kunszt PZ, Malik T, Raddick J, Stoughton C and vandenBerg J. The SDSS SkyServer Public access to the sloan digital sky server data. *International Conference on Management of Data*, pages 570 – 581 (2002).

Tai K, Baaden M, Murdock S, Wu B, Ng M, Johnston S, Boardman R, Fangohr H, Cox K, Essex JW and Sansom MSP. Three hydrolases and a transferase: comparative analysis of active-site dynamics via the BioSimGrid database. *Journal of Molecular Graphics and Modelling* (2006).

Tai K, Murdock S, Wu B, Ng M, Johnston S, Fangohr H, Cox SJ, Jeffreys P, Essex JW and Sansom MSP. BioSimGrid: Towards a worldwide repository for biomolecular simulations. *Organic & Biomolecular Chemistry*, **2**, 3219–3221 (2004).

Thompson DA and Best JS. The future of magnetic data storage technology. *IBM J. RES. DEVELOP.*, **44**(3) (2000).

Tiller J. *A Technical Guide to IPSec Virtual Private Networks*, chapter Reading and Writing to Files and Streams. Auerbach, 1^{st} edition (2000).

Tuomi I. The live and death of moore's law. *First Monday*, **7**(11) (2002). [Online; accessed 21-December-2005], `http://firstmonday.org/issues/issue7_11/tuomi/index.html`.

Tweedie S. Ext3, journaling file system. *Ottawa Linux Symposium, Ottawa Congress Center* (2000).

Vadala D. *Managing RAID on Linux*, chapter 2, pages 31–32. O'Reilly Media, Inc., 1^{st} edition edition (2002a).

Vadala D. *Managing RAID on Linux*. O'Reilly Media, Inc., 1^{st} edition edition (2002b).

Versant. Objects end-to-end: The ODBMS advantage (2001). [Online; accessed 1-August-2004], `http://www.versant.com`.

W3C. Document Object Model (DOM) level 1 specification (1998). [Online; accessed 1-January-2006], `www.w3.org/TR/1998/REC-DOM-Level-1-19981001`.

W3C. Namespaces in XML. *World Wide Web Consortium* (1999). REC-xml-names-19990114, [Online; accessed 21-February-2006], `www.w3.org/TR/1999/REC-xml-names-19990114`.

W3C. XML path language (XPath),version 1.0 (1999). [Online; accessed 1-August-2004], www.w3.org/TR/xpath.

W3C. Web Services Description Language (WSDL) 1.1 (2001). [Online; accessed 23-January-2006], www.w3.org/TR/wsdl.

W3C. SOAP Version 1.2 Part 1: Messaging Framework (W3C Recommendation) (2003). [Online; accessed 1-June-2004], www.w3.org/TR/soap12-part1/.

W3C. Extensible markup language (XML) 1.0 (2004a). [Online; accessed 1-June-2004], www.w3c.org/xml.

W3C. Resource Description Framework (RDF) schema specification. *W3C technical reports* (2004b). [Online; accessed 12-June-2006], www.w3.org/TR/PR-rdf-schema.

W3C. SOAP: Message Transmission Optimization Mechanism (2005a). [Online; accessed 1-January-2005], www.w3.org/TR/soap12-mtom/.

W3C. XML-binary Optimized Packaging (2005b). [Online; accessed 1-January-2005], www.w3.org/TR/xop10/.

W3C. XQuery 1.0: An XML query language, W3C working draft (2005c). [Online; accessed 1-August-2004], www.w3.org/TR/2005/WD-xquery-20050915/.

Wan M, Rajasekar A and Schroeder W. An Overview of the SRB 3.0: the Federated MCAT (2003). [Online; accessed 1-January-2005], www.sdsc.edu/srb/FedMcat.html.

Weiner PK and Kollman PA. AMBER: assisted model building with energy refinement. a general program for modeling molecules and their interactions. *J. Comp. Chem*, **2** (1981).

Wilkinson K, Sayers C, Kuno H and Reynolds D. Efficient RDF storage and retrieval in Jena2. *First International Workshop on Semantic Web and Databases*, (2003).

William J. Bolosky and SC, Goebel D and Douceur JR. Single instance storage in Windows 2000. *Microsoft Research* (2000).

Woods CJ, Ng M, Johnston S, Murdock SE, Wu B, Tai K, Fangohr H, Jeffreys P, Cox S, Frey JG, Sansom MSP and Essex JW. Grid computing and biomolecular simulation. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, **363**(1833) (2005).

Wu B, Dovey M, Ng M, Tai K, Murdock S, Fangohr H, Johnston S, Jeffreys P, Cox S, Essex J and Sansom MSP. A web / Grid portal implementation of BioSimGrid: A biomolecular simulation database. *Journal of Digital Information Management*, **2**(2), 74–78 (2004a).

Wu B, Dovey M, Tai K, Ng M, Stuart, Murdock, Fangohr H, Johnston S, Jeffreys P, Cox S, Essex JW and Sansom MS. Security and BioSimGrid: A biomolecular simulation database. *Proceedings of Workshop on Grid Security Practice and Experience* (2004b). Published as: University of York, Department of Computer Science Technical Report YCS-2004-380.

Wu B, Tai K, Murdock S, Ng M, Johnston S, Fangohr H, Jeffreys P, Cox S, Essex J and Sansom MS. BioSimGrid: A distributed database for biomolecular simulations. *Proceedings of UK e-Science All Hands Meeting 2003*, pages 412–419 (2003).

Wu B, Tai K, Ng M, Johnston S, Murdock S, Fangohr H, Sansom MSP, Essex J, Jeffreys P and Cox S. Towards a Grid-enabled biomolecular simulation database. *In Proceedings of UK e-Science All Hands Meeting 2005*, pages 577–580 (2005).

Xue G, Pound GE and Cox SJ. *Performing Grid Computation with Enhanced Web Service and Service Invocation Technologies*, volume 2659, pages 297–306. Springer Berlin / Heidelberg (2003). Lecture Notes in Computer Science.

Yergeau F. *UTF-8, a transformation format of ISO 10646*. The Internet Society (2003). RFC3629.