THE UNIVERSITY OF
MELBOURNE

Storage Exchange: A Global Platform for Trading Distributed Storage Services

by

Martin Placek

Submitted in total fulfilment of
the requirements for the degree of

Master of Engineering Science

Department of Computer Science and Software Engineering
The University of Melbourne
Australia

July, 2006

Storage Exchange: A Global Platform for Trading Distributed Storage Services

Martin Placek

Supervisor: Dr Rajkumar Buyya

ABSTRACT

The Storage Exchange is a new platform allowing storage to be treated as a tradeable resource. Organisations with varying storage requirements can use the Storage Exchange platform to trade and exchange storage services. Organisations have the ability to federate their storage, be-it dedicated or scavenged and advertise it to a global storage market.

This thesis provides a detailed account of the Storage Exchange and presents three main contributions in the field of distributed storage and the process required to realise a global storage utility. The first is a taxonomy of distributed storage systems covering a wide array of topics from the past and present. The second contribution involves proposing and developing the Storage Exchange, a global trading platform for distributed storage services. The development of the Storage Exchange platform identifies challenges and the necessary work required to make the global trading and sharing of distributed storage services possible.

The third and final contribution consists of proposing and evaluating Double Auction clearing algorithms which allow goods with indivisible demand constraints to be allocated in polynomial time. The process of optimally clearing goods of this nature in a Double Auction normally requires solving an NP-hard problem and is thus considered computationally intractable.

This is to certify that

(i) the thesis comprises only my original work,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is less than 30,000 words in length, exclusive of tables, maps, bibliographies, appendices and footnotes.

Martin Placek

July 2006

ACKNOWLEDGMENTS

The work described in this thesis was conducted with the assistance and support of many people to whom I would like to express my thanks. The most notable influence on my research activities has been my supervisor Dr Rajkumar Buyya. I'd also like to thank Professor Kotagiri Ramamohanarao and Dr Shanika Karunasekera who as committee members challenged the way I viewed my research, helping me to broaden my approach and adopt a different perspective when faced with problems which otherwise seemed insurmountable. I would like to thank Dr Charles Milligan and the Storage Technology Corporation (StorageTek) as early discussions with Charles provided the seed with which this research began. I would also like to thank Dr Jayant Kalagnanam (IBM T.J. Watson Research Center) for his time in answering my questions regarding his work on complexity analysis of double auctions.

My family have always been a significant influence. Their understanding and encouragement provided me with the confidence to pursue postgraduate work above other considerations. Their position made me focus on giving my best efforts during this candidature.

I would also like to collectively thank the members of the Department of Computer Science and Software Engineering. All the staff and students with whom I have had contact have always been very cordial and engaging. They helped to provide an environment where one is comfortable, care free and able to unobtrusively work towards an objective, all of this I have found invaluable.

I'd like to especially thank Thomas Manoukian for taking the time out of his busy schedule to provide me with feedback in many aspects of my research. I'd also like to thank the many people at Ceroc and City Salsa dance studios, especially Stephanie, Jen, Damien, Amy, Nadia and Peter who have listened to all my tales of research adventures and provided me with much encouragement throughout my

research candidature. They have helped me to enjoy my time outside of research and allow me to come back with a fresh outlook on things. Last, but by no means the least, Dr Anthony Senyard, who started my interest in research during my time as an undergraduate student.

This work was supported by an Australian Postgraduate Award (APA) and NICTA Victoria Laboratory whom I'd like to both thank whole-heartedly for making this experience possible. I'd also like to thank StorageTek for sponsoring the Grid computing Fellowship at the University of Melbourne, which was held by my supervisor from 2004-2005.

Martin Placek
Melbourne, Australia
July 2006.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

This chapter begins by introducing areas of research relevant to the work presented in this thesis. It discusses how aspects of distributed storage, grid computing and autonomic computing intersect and form the basis for the Storage Exchange, a globally distributed storage trading platform. This is followed by a discussion of the underlying motivating factors and primary contributions made. This chapter concludes with a discussion on the organisation of the remainder of the thesis.

1.1 Background Research

Storage plays a fundamental role in computing, a key element, ever present from registers and RAM to hard-drives and optical drives. Functionally, storage may service a range of requirements, from caching (expensive, volatile and fast) to archival (inexpensive, persistent and slow). Combining storage with networking has created a platform with endless possibilities allowing Distributed Storage Systems (DSSs) to adopt vast and varied roles, well beyond data storage.

Networking infrastructure and distributed storage systems share a close relationship. Advances in networking are typically followed by new distributed storage systems, which better utilise the network's capability. To illustrate, when networks evolved from primarily being private Local Area Networks (LANs) to public global Wide Area Networks (WANs) such as the Internet, a whole new generation of DSSs emerged, capable of servicing a global audience. The Internet has proven to be a source of many exciting and innovative applications and has enabled users to share and exchange resources across geographic boundaries. Terms such as pervasive, ubiquitous and federate were coined and heralded the rise of Grid Computing [108], which focuses on addressing the challenges associated with coordinating and sharing

heterogeneous resources across multiple geographic and administrative domains [53]. One of these challenges is data management, whose requirements led to the Data Grid [22]. Other issues concerning managing globally distributed data include providing a standard uniform interface across a heterogeneous set of systems [106], coordinating and processing of data [144] and managing necessary meta-data [73].

Distributed systems designed to successfully operate on the Internet are faced with many obstacles such as longer delays, unreliability, unpredictable and potentially malicious behaviour, associated with operating in a public shared environment. To cope with this, innovative architectures and algorithms have been proposed and developed, providing a stream of improvements to security, consistency and routing. As systems continue to advance, they increase in complexity and the expertise required to operate them [72]. Unfortunately, the continuing increase in complexity is unsustainable and ultimately limited by human cognitive capacity [134]. To address this problem, the Autonomic Computing [80] initiative has emerged aiming to simplify and automate the management of large scale complex systems.

1.2 Autonomic Storage Management

Distributed Storage Systems are rapidly evolving into complex systems, requiring increasingly more resources to be spent on maintenance and administration. The problem has been recognised by industry, where as much as 90% spent of the storage budget is allocated to its management [136]. This makes distributed storage systems a primary candidate for Autonomic Computing, which can be used to simplify and reduce the effort spent on maintenance and administration. One way to autonomically manage resource allocation in computer systems is through the use of economic principles [15]. Based on these principles we propose a platform capable of trading and automatically allocating distributed storage services.

Let us imagine a global storage marketplace, allowing storage to be traded much like any other service. Consumers are able to purchase storage services without being concerned about the underlying complexities. From the consumer's perspective this greatly simplifies the process of acquiring storage services. A process that

involves selecting hardware, configuration and continuous maintenance is simplified to recognising a need for storage and setting a budget. The problem of finding a suitable storage service and maintenance becomes the platform's responsibility. The work presented in this thesis provides an important step towards realising this ideal and proposes the Storage Exchange platform.

The Storage Exchange allows distributed storage to be treated as a tradeable resource. Organisations with varying storage requirements can use the Storage Exchange platform to trade and exchange storage services. Organisations have the ability to federate their storage, be-it dedicated or scavenged and advertise it to a global storage market. The centre piece of the Storage Exchange is its market model, which is responsible for automatically allocating trades based upon consumer and provider requirements. We envisage the Storage Exchange platform could be further automated by extending brokers to apply multi-agent [122] principles to purchase or lease storage in an autonomic manner. The ultimate goal being a platform capable of autonomic management of distributed storage services.

1.3 Significance and Motivation

In this section we discuss the factors motivating our research and the significant possibilities which arise from realising the Storage Exchange. The Storage Exchange platform can be used in a collaborative manner, where participants exchange services for credits, or alternatively in an open marketplace where enterprises trade storage services. Whether in a collaborative or enterprise environment, the incentives for an organisation to use our Storage Exchange platform include:

1. *Monetary Gain:* Organisations providing storage services (*Providers*) are able to better utilise existing storage infrastructure in exchange for monetary gain. Organisations consuming these storage services (*Consumers*) have the ability to negotiate for storage services as they require them, without needing to incur the costs associated with purchasing and maintaining storage hardware.

2. *Common Objectives:* There may be organisations who may wish to exchange

storage services as they may have a mutual goal such as preservation of information [26].

3. *Spikes in Storage Requirements:* Research organisations may require temporary access to mass storage [143] (e.g. temporarily store data generated from scientific experiments) and in exchange may provide access to their storage services.

4. *Economies of Scale:* Consumers are able to acquire cheaper distributed storage services from providers dedicated to selling large quantities of storage, rather than building in-house storage solutions.

5. *Donate:* Organisations may wish to donate storage services, particularly if these services will assist a noble cause.

6. *Autonomic Storage Management:* Future brokers will trade based upon an organisation's storage requirements and budget, simplifying storage management.

The Storage Exchange is a dynamic platform which can be applied in many different ways whilst providing organisations with incentives to participate. This thesis discusses the design of the Storage Exchange, including an investigation of the Double Auction market model and a computationally practical clearing algorithm.

1.4 Contribution

This thesis makes three key contributions towards the understanding of distributed storage systems and by applying market principles, moves closer towards a storage utility. These include:

1. A taxonomy of distributed storage systems, discussing key topics affecting the design and development of distributed storage systems. Topics covered by the taxonomy include functionality, architecture, operating environment, usage patterns, autonomic management, federation, consistency and routing.

The taxonomy is followed by a survey of distributed storage systems serving to exemplify classifications made in our taxonomy. The taxonomy also identifies challenges facing distributed storage systems and relevant research.

2. The design and development of the Storage Exchange, an innovative platform allowing storage services to be traded across a global environment. Organisations with varying storage requirements can use the Storage Exchange platform to trade and exchange services. As a provider, an organisation has the ability to harness unused storage on their workstations and advertise it to a global market, better utilising their existing storage infrastructure. From a consumer's perspective, organisations seeking storage services can do so without incurring the initial expense and labour associated with maintaining their own storage infrastructure.

3. A set of unique clearing algorithms enabling goods with multiple attributes and divisible constraints to be cleared in polynomial time under a sealed Double Auction market model. The process of optimally clearing goods of this nature in a Double Auction model is computationally intractable, requiring solving an NP-hard optimisation problem [79]. Clearing algorithms proposed include Maximise Surplus, Optimise Utilisation and a hybrid scheme. These are incorporated into the Storage Exchange and evaluated through the use of simulations.

1.5 Thesis Organisation

The remainder of this thesis is organised as follows: Chapter 2 presents a taxonomy of distributed storage systems, including a survey of distributed storage systems which apply market principles to manage various facets of their operation. Chapter 3 is dedicated to the Storage Exchange, providing a system overview and details of the architecture and design. Chapter 4 introduces and compares various auction market models before presenting and evaluating Double Auction clearing algorithms, allowing goods with multiple attributes and divisible constraints to be cleared in

polynomial time. We conclude and present ideas for future work in Chapter 5.

Core chapters of this thesis are based upon a technical report and a conference paper, detailed below:

Chapter 2 is mostly derived from:

- Martin Placek and Rajkumar Buyya.
  A Taxonomy of Distributed Storage Systems, Technical Report, GRIDS-TR-2006-11, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, July 3, 2006.

Chapters 3 and 4 are partially derived from:

- Martin Placek and Rajkumar Buyya.
  Storage Exchange: A Global Trading Platform for Storage Services. In *Proceedings of the Twelfth European Conference on Parallel Computing, Euro-Par 2006, Dresden, Germany, 29 August - 1st September.*

Chapter 2

DISTRIBUTED STORAGE SYSTEMS

This chapter presents a taxonomy of key topics affecting research and development of distributed storage systems. The taxonomy shows distributed storage systems to offer a wide array of functionality, employ architectures with varying degrees of centralisation and operate across environments with varying trust and scalability. Furthermore, taxonomies on autonomic management, federation, consistency and routing provide an insight into the challenges faced by distributed storage systems and the research carried out to overcome them. The chapter continues by providing a survey of distributed storage systems which exemplify topics covered in the taxonomy. Our focus then shifts to surveying distributed storage systems which employ market models to manage various aspects of their operation. This chapter concludes by summarising our discussion of distributed storage systems, which leads to the proposal of the Storage Exchange, detailed in the next chapter.

2.1 Taxonomy of Distributed Storage Systems

We introduce each of the topics covered in our taxonomy and provide a brief insight into the relevant research findings:

1. *System Function (Section 2.1.1):* A classification of DSS functionality uncovers a wide array of behaviour, well beyond typical store and retrieve.

2. *Storage Architecture (Section 2.1.2):* We discuss various architectures employed by DSSs. Our investigation shows an evolution from centralised to the more recently favoured decentralised approach.

3. *Operating Environment (Section 2.1.3):* We identify various categories of operating environments and discuss how each influence design and architecture.

4. *Usage Patterns (Section 2.1.4):* A discussion and classification of various workloads experienced by DSSs. We observe that the operating environment has a major influence on usage patterns.

5. *Consistency (Section 2.1.5):* Distributing, replicating and supporting concurrent access are factors which challenge consistency. We discuss various approaches used to enforce consistency and the respective trade offs in performance, availability and choice of architecture.

6. *Security (Section 2.1.6):* With attention turning towards applications operating on the Internet, establishing a secure system is a challenging task which is made increasingly more difficult as DSSs adopt decentralised architectures. Our investigation covers traditional mechanisms as well as more recent approaches that have been developed for enforcing security in decentralised architectures.

7. *Autonomic Management (Section 2.1.7):* Systems are increasing in complexity at an unsustainable rate. Research into autonomic computing [80] aims to overcome this dilemma by automating and abstracting away system complexity, simplifying maintenance and administration.

8. *Federation (Section 2.1.8):* Many different formats and protocols are employed to store and access data, creating a difficult environment in which to share data and resources. Federation middleware aims to provide a single uniform homogeneous interface to what would otherwise be a heterogeneous cocktail of interfaces and protocols. Federation enables multiple institutions to share services, fostering collaboration whilst helping to reduce effort otherwise wasted on duplication.

9. *Routing and Network Overlays (Section 2.1.9):* This section discusses the various routing methods employed by distributed storage systems. In our investigation we find that the development of routing shares a close knit relationship with the architecture; from a static approach as employed by

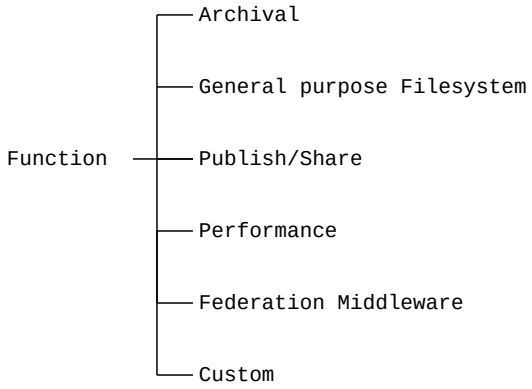

Figure 2.1: system function taxonomy

client-server architectures to a dynamic and evolving approach as employed by peer-to-peer.

2.1.1 System Function

In this section we identify categories of distributed storage systems (Figure 2.1). The categories are based on application functional requirements. We identify the following: (a) *Archival*, (b) *General purpose Filesystem*, (c) *Publish/Share*, (d) *Performance*, (e) *Federation Middleware* and (f) *Custom*.

Systems which fall under the archival category provide the user with the ability to backup and retrieve data. Consequently, their main objective is to provide persistent non-volatile storage. Achieving reliability, even in the event of failure, supersedes all other objectives and data replication is a key instrument in achieving this. Systems in this category are rarely required to make updates, their workloads follow a write-once and read-many pattern. Updates to an item are made possible by removing the old item and creating a new item and whilst this may seem inefficient, it is adequate for the expected workload. Having a write-once/read-many workload eliminates the likelihood of any inconsistencies arising due to concurrent updates, hence systems in this category either assume consistency or enforce a simple consistency model. Examples of storage systems in this category include PAST [46] and CFS [32].

Systems in the general purpose filesystem category aim to provide the user with persistent non-volatile storage with a filesystem like interface. This interface provides a layer of transparency to the user and applications which access it. The storage

behaves and thus complies to most, if not all, of the POSIX API standards [76] allowing existing applications to utilise storage without the need for modification or a re-build. Whilst systems in this category have ease of access advantage, enforcing the level of consistency required of a POSIX compliant filesystem is a non-trivial matter, often met with compromises. Systems which fall into this category include NFS [121], Coda [124, 123], xFS [4], Farsite [2] and Ivy [97].

Unlike the previous two categories where the storage service aims to be persistent, the publish/share category is somewhat volatile as the main objective is to provide a capability to share or publish files. The volatility of storage is usually dependent on the popularity of the file. This category of systems can be split into two further categories: (i) *Anonymity and Anti-censorship* and (ii) *File Sharing*. Systems in the anonymity and anti-censorship category focus on protecting user identity. While the storage is volatile, it has mechanisms to protect files from being censored. Systems in this category usually follow the strictest sense of peer-to-peer, avoiding any form of centralisation (discussed in greater detail in Section 2.1.2). Examples of systems which fall into this category include Free Haven [41], Freenet [24] and Publius [146]. The main objective for systems in the file sharing category is to provide the capability to share files amongst users. The system most famous for doing so, Napster [102], inspired the subsequent development of other systems in this category; Gnutella [102], MojoNation [151] and Bittorrent [68] to name a few.

DSSs in the performance category are typically used by applications which require a high level of performance. A large proportion of systems in this category would be classified as Parallel File Systems (PFSs). PFSs typically operate within a computer cluster, satisfying storage requirements of large I/O-intensive parallel applications. Clusters comprise of nodes interconnected by a high bandwidth and low latency network (e.g. Myrinet). These systems typically stripe data across multiple nodes to aggregate bandwidth. It is common for systems in this category to achieve speeds in the GB/sec bracket, speeds unattainable by other categories of DSSs. Commercial systems use fibre channel or iSCSI to connect storage nodes together to create a Storage Area Network (SAN), providing a high performance storage service. To best utilise the high performance potential, DSSs in this

category are specifically tuned to the application workload and provide an interface which mimics a general purpose filesystem interface. However, a custom interface (e.g. MPI-IO) that is more suited to parallel application development may also be adopted. Systems which fall into this category include PPFS [77], Zebra [67], PVFS [16], Lustre [13], GPFS [126], Frangipani [140], PIOUS [96] and Galley [99].

Global connectivity offered by the Internet allows institutions to integrate vast arrays of storage systems. As each storage system has varying capabilities and interfaces, the development of federation middleware is required to make interoperation possible in a heterogeneous environment. Middleware in this category is discussed in greater detail in Section 2.1.8. Systems which fall into this category are not directly responsible for storing data, instead they are responsible for high-level objectives such as cross domain security, providing a homogeneous interface, managing replicas and the processing of data. Generally speaking, much of the research into Data Grids [22, 73, 144, 7] is relevant to federation middleware.

Finally, the custom category has been created for storage systems that possess a unique set of functional requirements. Systems in this category may fit into a combination of the above system categories and exhibit unique behaviour. Google File System (GFS) [57] and OceanStore [82, 109], are examples of such systems. GFS has been built with a particular functional purpose which is reflected in its design (Section 2.2.7). OceanStore aims to be a global storage utility, providing many interfaces including a general purpose filesystem. To ensure scalability and resilience in the event of failure, OceanStore employs peer-to-peer mechanisms to distribute and archive data. Freeloader [143] combines storage scavenging and striping, achieving good parallel bandwidth on shared resources. The array of features offered by Freeloader, OceanStore and the purpose built GFS all exhibit unique qualities and are consequently classified as custom.

2.1.2 Storage Architecture

In this section our focus turns to distributed storage system architectures. The architecture determines the application's operational boundaries, ultimately forging behaviour and functionality. There are two main categories of architectures (Figure

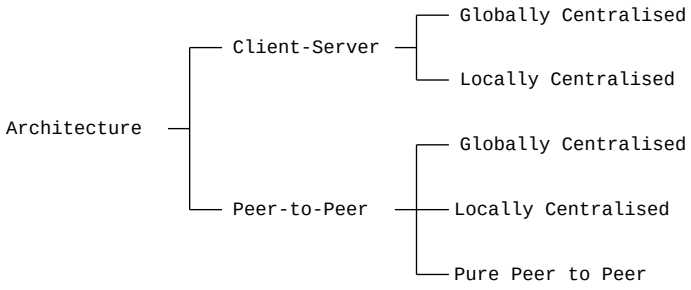

Figure 2.2: architecture taxonomy

2.2), *client-server* and *peer-to-peer.* The roles which an entity may embrace within a client-server architecture are very clear, an entity may exclusively behave as either a client or a server, but cannot be both [128]. On the contrary, participants within a peer-to-peer architecture may adopt both a client and a server role. A peer-to-peer architecture in its strictest sense is completely symmetrical, each entity is as capable as the next. The rest of this section discusses both categories in greater detail.

A client-server based architecture revolves around the server providing a service to requesting clients. This architecture has been widely adopted by distributed storage systems past and present [121, 4, 95, 124, 140, 143, 57]. In a client-server architecture, there is no ambiguity concerning who is in control, the server is the central point, responsible for authentication, consistency, replication, backup and servicing requesting clients. A client-server architecture may exhibit varying levels of centralisation and we have identified two categories *Globally Centralised* and *Locally Centralised*. A globally centralised architecture contains a single central entity being the server, this results in a highly centralised architecture which has limited scalability and is susceptible to failure. To alleviate problems associated with a single central server, a locally centralised architecture distributes responsibilities across multiple servers allowing these systems [4, 124, 140, 143, 57] to be more resilient to outages, scale better and aggregate performance. However, even a locally centralised architecture is inherently centralised, making it vulnerable to failure and scalability bottlenecks. A client-server architecture is suited to a controlled environment, either trusted or partially trusted (Section 2.1.3). Operating in a controlled environment allows the focus to shift to performance, strong consistency and providing a POSIX file I/O interface.

To meet the challenges associated with operating in an ad-hoc untrusted environment such as the Internet, a new generation of systems adopting a peer-to-peer architecture have emerged. In a peer-to-peer architecture every node has the potential to behave as a server and a client, and join and leave as they wish. Routing continually adapts to what is an ever changing environment. Strengths of the peer-to-peer approach include resilience to outages, high scalability and an ability to service an unrestricted public user-base. These strengths vary depending on the category of peer-to-peer a system adopts.

There are three main categories of peer-to-peer architectures, *Globally Centralised, Locally Centralised* and *Pure Peer-to-Peer.* Each of these categories have a varying degree of centralisation, from being globally centralised to locally centralised to having little or no centralisation with pure peer-to-peer. One of the early peer-to-peer publishing packages, Napster [102] is an example of a system employing a globally centralised architecture. Here, peers are required to contact a central server containing details of other peers and respective files. Unfortunately, this reliance on a globally central index server limits scalability and proves to be a Single Point of Failure (SPF).

Locally centralised architectures were inspired by the shortcomings of early peer-to-peer efforts. Gnutella [102] initially relied on broadcasting to relay queries although this proved to be a bottleneck, with as much as 50% [30] of the traffic attributed to queries. To overcome this scalability bottleneck, a locally centralised architecture employs a select few hosts with high performance and reliable characteristics to behave as *super nodes.* These super nodes maintain a repository of meta-data which a community of local nodes may query and update. Super nodes communicate amongst each other forming bridges between communities, allowing local nodes to submit queries to a super node rather than broadcasting to the entire community. Whilst super nodes introduce an element of centralisation, in sufficient numbers, they avoid becoming points of failure. Examples of peer-to-peer systems which use a locally centralised architecture include FastTrack [39, 68], Clippee [3], Bittorrent [68] and eDonkey [142].

Without any central entities, a pure peer-to-peer architecture exhibits symmet-

rical harmony between all entities. The symmetrical nature ensures that it is the most scalable of the three and proves to be very capable at adapting to a dynamic environment. Whilst on the surface it may seem that this is the architecture of choice, adhering to a pure peer-to-peer philosophy is challenging. Achieving a symmetrical relationship between nodes is made difficult in the presence an asymmetric [102] network such as the Internet. User connections on the Internet are usually biased towards downloads, sometimes by as much as 600% (1.5Mb down/256Kb up). This bias discourages users from sharing their resource, which in turn hinders the quality of service provided by the peer-to-peer system. The way in which nodes join [151] a peer-to-peer network also poses a challenge. For example, if every node were to join the network through one node, this would introduce a SPF, something which peer-to-peer networks need to avoid. Finally the lack of centralisation and the ad-hoc environment a peer-to-peer system operates in makes establishing trust and accountability essential but difficult to do without ironically introducing some level of centralisation or neighbourhood knowledge. Systems which closely follow a pure peer-to-peer architecture include Free Haven [41], Freenet [24] and Ivy [97].

The choice of architecture has a major influence on system functionality, determining operational boundaries and its effectiveness to operate in a particular environment. As well as functional aspects, the architecture also has a bearing on the mechanisms a system may employ to achieve consistency, routing and security. A centralised architecture is suited to controlled environments and while it may lack the scalability of its peer-to-peer counterpart, it has the ability to provide a consistent Quality of Service (QoS). By contrast a peer-to-peer architecture is naturally suited to a dynamic environment, key advantages include unparalleled scalability and the ability to adapt to a dynamic operating environment. Our discussion of architectures in this section has been presented in a chronological order. We can see that the evolution of architectures adopted by DSSs have gradually moved away from centralised to more decentralised approaches (Figure 2.3), adapting to challenges associated with operating across a dynamic global network.

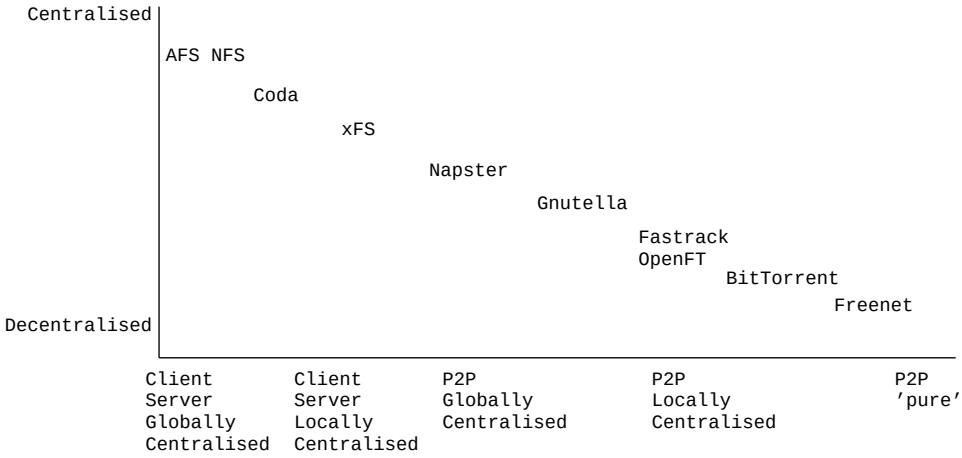

Figure 2.3: architecture evolution

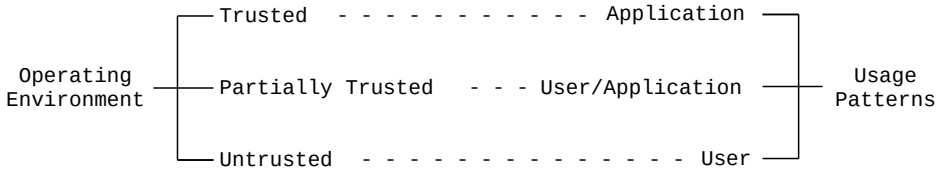

Figure 2.4: operating environment and usage taxonomy

2.1.3 Operating Environment

This section discusses the possible target environments which distributed storage systems may operate in. While examining each operating environment, a discussion of the influence on architecture and the resulting workload is made. We have identified three main types of environments, (a) *Trusted*, (b) *Partially Trusted* and (c) *Untrusted*, as shown in Figure 2.4.

A trusted environment is dedicated and quarantined off from other networks. This makes the environment very controlled and predictable. Users are restricted and therefore accountable. Its controlled nature ensures a high level of QoS and trust, although in general scalability is limited. Administration is carried out under a common domain and therefore security is simpler compared to environments that stretch beyond the boundaries of an institution. Due to the controlled nature of a trusted environment, workload analysis can be conducted without the need to consider the unpredictable behaviour exhibited by external entities. As the workload is primarily influenced by the application, the storage system can be optimised to suit the workload. As the storage system's main goal is performance, less emphasis

is given to adhering to the standard POSIX File I/O Interface [76]. A cluster is a good example of a trusted environment.

Distributed storage systems which operate in a partially trusted environment are exposed to a combination of trusted and untrusted nodes. These nodes operate within the bounds of an organisation. The user base is also limited to the personnel within the organisation. Whilst a level of trust can be assumed, security must accommodate for "the enemy within" [130, 119]. This environment is not as controlled as a trusted environment as many other applications may need to share the same resources and as such this introduces a level of unpredictability. As the network is a shared resource, DSSs need to utilise it conscientiously so as not to impede other users. In a partially trusted environment, DSSs are primarily designed for maximum compatibility and the provision of a general purpose filesystem interface.

In an untrusted environment, every aspect (nodes and network infrastructure) is untrusted and open to the public. An environment which best likens itself to this is the Internet. In an open environment where accountability is difficult if not impossible [102], a system can be subjected to a multitude of attacks [40]. With the emergence of peer-to-peer systems allowing every host to be as capable as the next, it is important to understand user behaviour and the possible perils. Some lessons learnt include a very transient user base (also referred to as churn) [151], *tragedy of the commons* [64] and the *Slashdot effect* [1].

Early research [125, 133] discusses issues associated with scaling up client-server distributed storage systems (Andrew [95] and Coda [124]) across a WAN. Some of the problems identified include (i) a lower level of trust between users, (ii) coordination of administration is difficult, (iii) network performance is degraded and failures are more common than what is found in a LAN environment. DSSs need to overcome challenging constraints imposed by an untrusted environment. Achieving a robust and secure storage service whilst operating in an untrusted environment is a source of much ongoing research.

Our survey of DSSs has found the operating environment has a major influence on system design and the predictability of workload. A trusted environment has the advantage of being sheltered from the unpredictable entities otherwise present in

partially trusted and untrusted environments. The predictability and controlled nature of a trusted environment is suitable for a client-server architecture. In contrast, the dynamic nature of a partially trusted or untrusted environment requires that a more ad-hoc approach to architecture be employed, such as peer-to-peer.

2.1.4 Usage Patterns

Collection and analysis of usage data including various file operations and attributes plays an important role in the design and tuning of DSSs. Empirical studies serve to provide an important insight into usage trends, identifying possible challenges and the necessary research to overcome them. In our investigation, we found usage patterns to be closely related to the operating environment (Figure 2.4) and for this reason our discussion of usage patterns is organised based on operating environments. This section summarises empirical studies based on DSSs which operate in a partially trusted environment, a trusted environment and finally in an untrusted environment.

Partially Trusted

A study [100] focusing on the usage patterns of the Coda [124] (Section 2.2.4) storage system makes some interesting observations regarding file usage whilst disconnected from the file server. Coda employs an optimistic approach to consistency (Section 2.1.5) permitting users to continue to work on locally cached files even without network connectivity. During the study, the authors found there to be a surprisingly high occurrence of *integration failures* or change conflicts. A change conflict occurs when a user reconnects to merge their changes with files that have already been modified during the period the user was disconnected. A file server attempting to merge a conflicting change will fail to do so, requiring the users to merge their changes manually. Whilst some of these change conflicts were due to servers disappearing during the process of merging changes, there still remained a high proportion of conflicts. This occurrence suggested that disconnected users do not work on widely distinct files as previously thought, this is an important realisation for DSSs adopting an optimistic approach to consistency.

A survey [45] conducted across 4800 workstations within Microsoft found only half of the filesystem storage capacity to be utilised. These results inspired a subsequent feasibility study [12] on accessing this untapped storage. The feasibility study focused on machine availability, filesystem measurements and machine load. The results supported earlier findings with only 53% of disk space being used, half of the machines remained available for over 95% of the time, machine's cpu load average to be 18% and 70% of the time the machine's disks were idle. The results of the feasibility study found that developing a storage system which utilised available storage from shared workstations was in fact possible and consequently lead to the development of Farsite [2] (Section 2.2.3).

A number of other empirical studies relevant to the partially trusted category include: A comparatively early study [133] primarily focusing on the use of AFS [74] whilst another study adopted a developer's perspective [58], analysing source code and object file attributes. That study found more read-write sharing to be present in an industrial environment than typically found in an academic environment. DSSs operating in a partially trusted environment aim to provide an all-purpose solution, servicing a wide array of applications and users. Due to the general nature of these storage systems, studies analysing usage patterns are influenced by a combination of user and application behaviour.

Untrusted

Usage patterns of applications designed to operate in an untrusted environment are primarily influenced by user behaviour. Applications which adopt a peer-to-peer approach serve as primary examples, empowering every user with the ability to provide a service. With these type of systems, it is therefore important to understand user behaviour and the resulting consequences. Past experience from deploying MojoNation [151] show how flash crowds have the ability to cripple a system with any element of centralisation in its architecture. When MojoNation was publicised on Slashdot, their downloads skyrocketed from 300 to 10,000 a day. Even though MojoNation employs a peer-to-peer architecture for its day-to-day operation, a central server assigned to handling new MojoNation users was overwhelmed,

rendering it unavailable. Further observations include: a very transient user base with 80% to 84% of users being connected once and for less than an hour and users with high-bandwidth and highly-available resources being least likely to stay connected for considerable lengths of time.

Systems adopting a peer-to-peer philosophy rely on users cooperating and sharing their services, unfortunately there are many disincentives [51] resulting in peer-to-peer systems being vulnerable to freeriding, where users mainly consume services without providing any in return. Studies show that [102, 51, 75] the primary reason for this behaviour is due to the asymmetrical nature of users' connections, being very biased towards downloading. A usage study of Gnutella [75] found that 85% of users were freeriding. To discourage this and promote cooperation, the next generation of peer-to-peer systems (Maze [153], Bittorrent [10]) provide incentives for users to contribute services.

Trusted

Unlike the previous two categories, storage systems operating in a trusted environment (e.g. clusters) service a workload primarily influenced by application behaviour. A trusted environment is dedicated, making it predictable and controlled, eliminating variables otherwise found in shared environments, leaving the application as the main influence of usage patterns. Currently the vastly superior performance of CPU and memory over network infrastructure has resulted in networking being the bottleneck for many parallel applications, especially if heavily reliant on storage. Hence, understanding the application's workload and tuning the storage system to suit plays an important role in improving storage performance, reducing the network bottleneck and realising a system running closer to its full potential. A usage pattern study [28] of various parallel applications found that each application had its own unique access pattern. The study concluded that understanding an application's access pattern and tuning the storage system (caching and prefetching) to suite was the key to realising the full potential of parallel filesystems.

The Google File System (GFS) [57] (Section 2.2.7) is another example highlighting the importance of understanding an application's usage pattern and the

advantages of designing a storage system accordingly. The authors of the GFS made a number of key observations on the type of workload their storage system would need to service and consequently designed the system to accommodate this. The GFS typical file size was expected to be in the order of GB's and the application workload would consist of large continuous reads and writes. Based on this workload, they adopted a relaxed consistency model with a large chunk size of 64MB. Choosing a large chunk size proved beneficial as (i) the client spent less time issuing chunk lookup requests, (ii) the meta-data server had less chunk requests to process and consequently chunk entries to store and manage.

2.1.5 Consistency

The emergence and subsequent wide proliferation of the Internet and mobile computing has been a paradox of sorts. Whilst networks are becoming increasingly pervasive, the connectivity offered is unreliable, unpredictable and uncontrollable. The result effect is a network that imposes challenging operational constraints on distributed applications. More specific to storage systems, the Internet and mobile computing increase availability and the risk of concurrent access and unexpected outages have the potential to partition networks, further challenging data consistency. This section discusses various mechanisms employed by DSSs to ensure data remains consistent even in the presence of events which challenge it. Our discussion of consistency begins from a database viewpoint outlining principles and terminology and continues with a discussion of various approaches storage systems employ.

Principles and Terminology

In this section we shall cover the underlying principles and terminology relevant to consistency. The topic has received much attention in the area of transactions and databases and thus we shall draw upon these works [62, 34] to provide a brief introduction. Whilst terminology used to describe consistency in databases (transactions and tables) may differ to DSSs (file operations and files) the concepts

are universal. Consistency ensures the state of the system remains consistent or correct even when faced with events (e.g. concurrent writers, outages) which would otherwise result in an inconsistent or corrupt state. The ACID principles, serializability, levels of isolation and locking are all important terms which lay the foundations for consistency and we shall now discuss each briefly.

The ACID (Atomic, Consistent, Isolation, Durability) [63] principles describe a set of axioms, that if enforced, will ensure the system remains in a consistent state. A system is deemed to uphold ACID principles if:

1. ***A****tomic:* Transaction is atomic, that is, all changes are completed or none are. *(all or nothing).*

2. ***C****onsistency:* Transactions preserve consistency. Assuming a database is in a consistent state to begin with, a transaction must ensure that upon completion the database remains in a consistent state.

3. ***I****solation:* Operations performed within the life-cycle of a transaction must be performed independently and unbeknown to other transactions running concurrently. The strictest sense of isolation is referred to as serializability (see below). A system may guarantee varying degrees of isolation each with their trade-offs.

4. ***D****urablity:* Once a transaction has completed the system must guarantee that any modifications done are permanent even in the face of subsequent failures.

Serializability is a term use to describe a *criterion of correctness*. A set of transactions is deemed serializable if their result is *some* serial execution of the same transactions. In other words, even though the execution of these transactions may have been interleaved, as long as the final result is achieved by some serial order of execution, their execution is deemed serializable and thus correct. To guarantee a transaction is serializable, its execution needs to adhere to the two-phase locking [49] theorem. The theorem outlines the following two axioms on acquiring and releasing locks:

1. Before executing any operations on data, a transaction must acquire a lock on that object.

2. Upon releasing a lock, the transaction must not acquire any more locks.

Serializability achieves maximum isolation, with no interference allowed amongst executing transactions. The ANSI/ISO SQL standard (SQL92) identifies four degrees of isolation. To offer varying levels of isolation, transactions may violate the two-phase locking theorem and release locks early and acquire new locks. Violating the two-phase locking protocol relaxes the degree of isolation allowing for a greater level of concurrency and performance at the cost of correctness. The SQL standard identifies the following three possible ways in which serializability may be violated:

1. *Dirty Read:* Uncommitted modifications are visible by other transactions. Transaction A inserts a record, Transaction B is able to read the record, Transaction A than executes a rollback leaving Transaction B with a record which no longer exists.

2. *Non-Repeatable Read:* Subsequent reads may return modified records. Transaction A executes a query on table A. Transaction B may insert, update and delete records in table A. Assuming Transaction B has committed its changes, when Transaction A repeats the query on Table A changes made by Transaction B will be visible.

3. *Phantom Read:* Subsequent read may return additional (phantom) records. Transaction A executes a query on table A. Transaction B than inserts a record into the table A and commits. Transaction A then executes, repeats its original query of table A and finds an additional record.

Therefore a database typically supports the following four levels of consistency, with repeatable read usually being the default:

1. *Serializability:* To achieve serializability, transactions executing concurrently must execute in complete isolation and must not interfere with each other. Transactions must adhere to the two-phase locking protocol to achieve

serializability. Whilst this offers the highest level of isolation possible, a subsequent penalty is poor concurrency.

2. *Repeatable Read:* Repeatable read ensures that data retrieved from an earlier query will not be changed for the life of the transaction. Therefore subsequent executions of the same query will always return the same records unmodified, although additional (phantom) records are possible. Repeatable Read employs shared read locks which only covers existing data queried. Other transactions are allowed to 'insert' records giving rise to *Phantom Reads*.

3. *Read Committed:* Transactions release read locks early (upon completion of read), allowing other transactions to make changes to data. When a transaction repeats a read, it reacquires a read lock although results may have been modified, resulting in a *Non-Repeatable Read*.

4. *Read Uncommitted:* Write locks are released early (upon completion of write), allowing modifications to be immediately visible by other transactions. As data is made visible before it has been committed, other transactions are effectively performing *Dirty Reads* on data which may be rolled back.

Approaches

Thus far our discussion of consistency has been in the context of databases and transactions, which have been used to convey the general principles. There are two ways of approaching consistency, *Strong* or *Optimistic*, each method with its respective trade offs (Figure 2.5).

Strong consistency also known as pessimistic, ensures that data will always remain and be accessed in a consistent state, thus holding true to the ACID principles. A couple of methods which aim to achieve strong consistency include one copy serializability [8], locking and leasing. The main advantage of adopting a strong approach is that data will always remain in a consistent state. The disadvantages include limited concurrency and availability, resulting in a system with poor performance that is potentially complex if a distributed locking mechanism is employed. The other approach to consistency is optimistic consistency [83] which

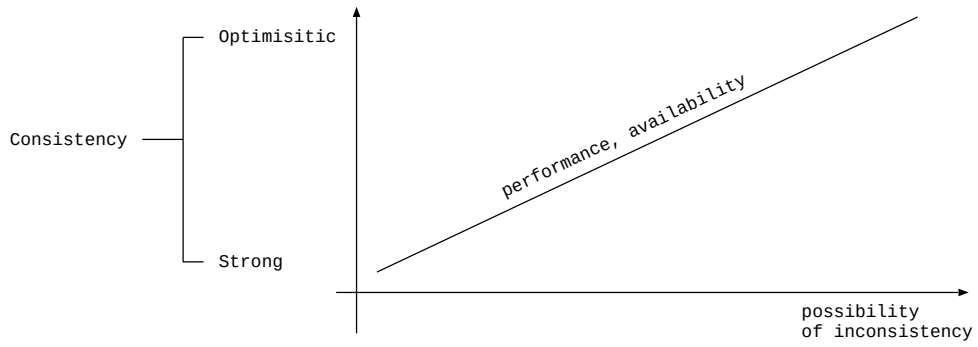

Figure 2.5: strong vs optimistic consistency

is also known as weak consistency. It is considered weak as it allows the system to operate whilst in an inconsistent state. Allowing concurrent access to partitioned replicas has the potential for inconsistent reads and modifications that fail to merge due to conflicting changes. The advantages associated with an optimistic approach include excellent concurrency, availability and consequently good scalable performance. The main drawbacks being inconsistent views and the risk of change conflicts which require user intervention to resolve.

Strong Consistency

The primary aim of strong consistency is to ensure data is viewed and always remains in a consistent state. To maintain strong consistency, locking mechanisms need to be employed. Put simply, a piece of data is locked to restrict user access. Much discussion and work [111, 112, 61] has gone into applying locks and choosing an appropriate grain size. Choosing a large grain to lock has the advantage of lowering the frequency at which locking be initiated, the disadvantages include increasing the probability of dealing with lock contention and low concurrency. Choosing a small grain to lock has the advantage of high concurrency, but carries an overhead associated with frequently acquiring locks. These grain size trade-offs are universal and also apply to a distributed storage environment.

In a distributed environment the performance penalty associated with employing a locking infrastructure is high. Distributed storage systems which support replication face the prospect of implementing a distributed locking service (Frangipani

[140] and OceanStore [82]) which incurs further performance penalties; a polynomial number of messages need to be exchanged between the group of machines using a Byzantine agreement (see Section 2.1.6). With these high overheads the choice to use a large block size is justified: e.g. 64MB used by the GFS [57]. However, careful analysis of storage workload is required as any performance gained from choosing a large block size would be annulled by the resulting lock contention otherwise present in a highly concurrent workload.

A locking infrastructure requires a central authority to manage and oversee lock requests. Therefore, DSSs choosing to employ locking to achieve consistency are restricted to architectures which contain varying degrees of centralisation (Table 2.1). A client-server architecture is ideal, leaving the server to be the central entity which enforces locking. Implementing a locking mechanism over a peer-to-peer architecture is a more involved process, which becomes impossible in a pure peer-to-peer architecture. Systems which choose to support strong consistency mostly operate in a partially trusted environment. The relatively controlled and reliable nature of a partially trusted environment suites the requirements imposed by strong consistency.

System	**Architecture**	**Environment**
Frangipani	Client-Server	Partially Trusted
NFS	Client-Server	Partially Trusted
Farsite	Locally Centralised Peer-to-Peer	Partially Trusted

Table 2.1: strong consistency - impact on architecture and environment

Optimistic Consistency

The primary purpose is to keep data consistent without imposing the restrictions associated with strong consistency. Optimistic consistency allows multiple readers and writers to work on data without the need for a central locking mechanism. Studies of storage workloads [81, 58] show that it is very rare for modifications to result in a change conflict and as such the measures used to enforce strong consistency are perceived as *overkill* and unnecessary. Taking an optimistic approach

to consistency is not unreasonable and in the rare event that a conflict should occur users will need to resolve conflicts manually.

An optimistic approach to consistency accommodates a dynamic environment, allowing for continuous operation even in the presence of partitioned replicas, this is particularly suited to unreliable connectivity of WANs (e.g. Internet). There are no limits imposed on the choice of architecture when adopting an optimistic approach and, as it is highly concurrent, it is well suited to a pure peer-to-peer architecture.

Examples of DSSs which employ an optimistic consistency model include: xFS [4], Coda [124] and Ivy [97]. Both Ivy and xFS employ a log structured filesystem, recording every filesystem operation into a log. By traversing the log it is possible to generate every version of the filesystem and if a change conflict arises it is possible to rollback to a consistent version. Coda allows the client to have a persistent cache, which enables the user to continue to function even when without a connection to the file server. Once a user reconnects, the client software will synchronize with the server's.

2.1.6 Security

Security is an integral part of DSSs, serving under many guises from authentication and data verification to anonymity and resilience to Denial-of-Service (DoS) attacks. In this section we shall discuss how system functionality (Section 2.1.1), architecture (Section 2.1.2) and operating environment (Section 2.1.3) all have an impact on security and the various methods (Figure 2.6) employed to enforce it. To illustrate, a storage system used to share public documents within a trusted environment need not enforce the level of security otherwise required by a system used to store sensitive information in an untrusted environment.

Systems which tend to operate within the confines of a single administration domain use ACL (Access Control List) to authenticate users and firewalls to restrict external access. These security methods are effective in controlled environments (partially trusted or trusted). Due to the controlled nature of these environments, the potential user base and hardware is restricted to within the bounds of an institution, allowing for some level of trust to be assumed. On the contrary,

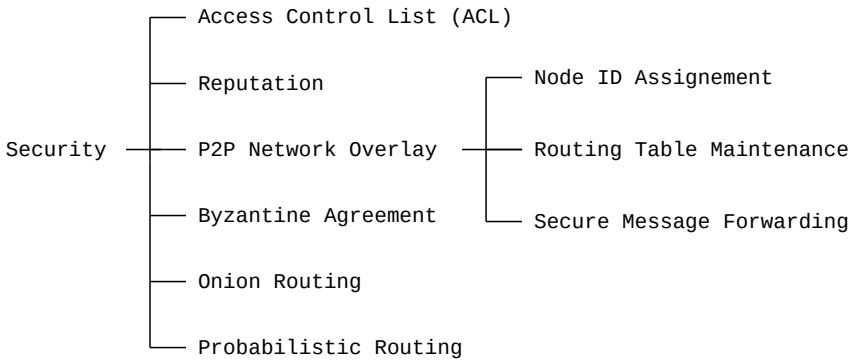

Figure 2.6: security taxonomy

untrusted environments such as the Internet expose systems to a global public user base, where any assumptions of trust are void. Storage systems which operate in an untrusted environment are exposed to a multitude of attacks [44, 40]. Defending against these is non-trivial and the source of much ongoing research.

The choice of architecture influences the methods used to defend against attacks. Architectures which accommodate a level of centralisation such as client-server or centralised peer-to-peer have the potential to either employ ACL or gather neighbourhood knowledge to establish reputations amongst an uncontrolled public user base. However, security methods applicable to a centralised architecture are inadequate in a pure peer-to-peer setting [66]. Systems adopting a pure peer-to-peer architecture have little, if any, element of centralisation and because of their autonomous nature are faced with further challenges in maintaining security [19, 131]. Current peer-to-peer systems employ network overlays (Section 2.1.9) as their means to communicate and query other hosts. Securing a peer-to-peer network overlay [19] decomposes into the following key factors:

1. *Node Id Assignment:* When a new node joins a peer-to-peer network it is assigned a random 128bit number which becomes the node's id. Allowing a node to assign itself an id is considered insecure making the system vulnerable to various attacks, including (i) attackers may assign themselves ids close to the document hash, allowing them to control access to the document, (ii) attackers may assign themselves ids contained in a user's routing table, effectively controlling that user's activities within the peer-to-peer network. Freenet [24]

attempts to overcome this problem by involving a chain of random nodes in the peer-to-peer network to prevent users from controlling node id selection. Assuming the user does not have control of node id selection, this still leaves the problem of users trying to dominate the network by obtaining a large number of node ids, this kind of attack is also known as the Sybil [44] attack. A centralised solution is proposed in [19], where a trusted entity is responsible for generating node ids and charging a fee to prevent the Sybil attack. Unfortunately this introduces centralisation and a SPF which ultimately could be used to control the peer-to-peer network itself.

2. *Routing Table Maintenance:* Every node within a peer-to-peer network overlay maintains a routing table that is dynamically updated as nodes join and leave the network. An attacker may attempt to influence routing tables, resulting in traffic being redirected through their faulty nodes. Network overlays which use proximity information to improve routing efficiency are particularly vulnerable to this type of attack. To avoid this, strong constraints need to be placed upon routing tables. By restricting route entries to only point to neighbours close in the node id space (CAN and Chord), attackers cannot use network proximity to influence routing tables. Whilst this results in a peer-to-peer network that is not susceptible to such an attack, it also disables any advantages gained from using network proximity based routing.

3. *Secure Message Forwarding:* All peer-to-peer network overlays provide a means of sending a message to a particular node. It is not uncommon for a message to be forwarded numerous times in the process of being routed to the target node. If any nodes along this route are faulty, this message will not reach the desired destination. A faulty node may choose not to pass on the message or pretend to be the destined node id. To overcome this, [19] proposes a failure test method to determine if a route works and suggests the use of a redundant routing path when this test fails.

The rest of this section discusses a few methods commonly used by DSSs to establish trust, enforce privacy, verify and protect data. A simple but effective way

of ensuring data validity is through the use of cryptographic hash functions such as the Secure Hash Algorithm (SHA) [98] or Message Digest algorithm (MD5) [113]. These algorithms calculate a unique hash which can be used to check data integrity. Due to the unique nature of the hash, distributed storage programs also use it as a unique identifier for that block of data. To protect data and provide confidentiality the use of the Public Key Infrastructure (PKI) allows data encryption and restricted access to audiences holding the correct keys.

The Byzantine agreement protocol [21] enables the establishment of trust within an untrusted environment. The algorithm is based on a voting scheme, where a Byzantine agreement is only possible when more than two thirds of participating nodes operate correctly. The protocol itself is quite network intensive with messages passed between nodes increasing in polynomial fashion with respect to the number of participants. Hence the number of participants which form a Byzantine group are limited and all require good connectivity. OceanStore [82] and Farsite [2] are both examples of systems which have successfully employed the Byzantine protocol to establish trust. Another way to establish trust is via a reputation scheme, rewarding good behaviour with credits and penalising bad behaviour. Free Haven [41] and MojoNation [151] use digital currency to encourage participating users to behave.

Systems such as Free Haven [41] and Freenet [24] both aim to provide users with anonymity and anti-censorship. These class of systems need to be resilient to many different attacks from potentially powerful adversaries whilst ensuring they do not compromise the very thing they were designed to protect. Introducing any degree of centralisation and neighbourhood intelligence into these systems is treated with caution [42, 88] as this makes the system vulnerable to attacks. Onion routing [102, 43, 137] and probabilistic routing [41] are two methods employed to provide anonymous and censorship resistant communications medium.

2.1.7 Autonomic Management

The evolution of DSSs has seen an improvement in availability, performance and resilience in the face of increasingly challenging constraints. To realise these improvements DSSs have grown to incorporate newer algorithms and more

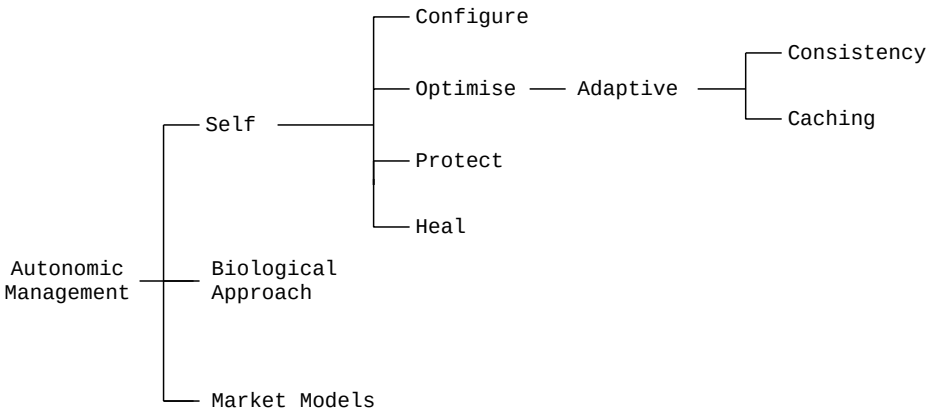

Figure 2.7: autonomic management taxonomy

components, increasing their complexity and the knowledge required to manage them. With this trend set to continue, research into addressing and managing complexity (Figure 2.7) has led to the emergence of autonomic computing [72, 80]. The autonomic computing initiative has identified the *complexity crisis* as a bottleneck, threatening to slow the continuous development of newer and more complex systems.

Distributed Storage Systems are no exception, evolving into large scale complex systems with a plethora of configurable attributes, making administration and management a daunting and error prone task [6]. To address this challenge, autonomic computing aims to simplify and automate the management of large scale complex systems. The autonomic computing vision, initially defined by eight characteristics [72], was later distilled into four [80]; self-configuration, self-optimisation, self-healing and self-protection, all of which fall under the umbrella of self management. We discuss each of the four aspects of autonomic behaviour and how they translate to autonomic storage in Table 2.2. Another approach to autonomic computing takes a more ad-hoc approach, drawing inspiration from biological models [134]. Both of these approaches are radical by nature, having broad long-term goals requiring many years of research to be fully realised. In the mean time, research [52, 152, 15, 149] with more immediate goals discuss the use of market models to autonomically manage resource allocation in computer systems. More specifically, examples of such storage systems and the market models employed

are listed below and discussed in greater detail in Section 2.3.

1. *Mungi [70]:* is a Single-Address-Space Operating System (SASOS) which employs a commodity market model to manage storage quota.

2. *Stanford Archival Repository Project [26]:* apply a bartering mechanism, where institutions barter amongst each other for distributed storage services for the purpose of archiving and preserving information.

3. *MojoNation [151]:* uses digital currency (*Mojo*) to encourage users to share and barter resources on its network, users which contribute are rewarded with *Mojo* which can be redeemed for services.

4. *OceanStore [82]:* is a globally scalable storage utility, providing paying users with a durable, highly available storage service by utilising untrusted infrastructure.

5. *Storage Exchange [104]:* applies a sealed Double Auction market model allowing institutions to trade distributed storage services. The Storage Exchange provides a framework for storage services to be brokered autonomically based on immediate requirements.

As distributed storage systems are continuing to evolve into grander, more complex systems, autonomic computing is set to play an important role, sheltering developers and administrators from the burdens associated with complexity.

2.1.8 Federation

Global connectivity provided by the Internet has allowed any host to communicate and interact with any other host. The capability for institutions to integrate systems, share resources and knowledge across institutional and geographic boundaries is available. Whilst the possibilities are endless, the middleware necessary to federate resources across institutional and geographic boundaries has sparked research in Grid computing [53]. Grid computing is faced with many challenges including: supporting cross domain administration, security, integration of heterogeneous systems, resource

1.**Self-configuration:** *Autonomic systems are configured with high-level policies, which translate to business-level objectives.* Large DSSs are governed by a myriad of configurable attributes, requiring experts to translate complex business rules into these configurables. Storage Policies [37] provide a means by which high-level objectives can be defined. The autonomic component is responsible for translating these high-level objectives into low level configurables, simplifying the process of configuration.
2.**Self-optimisation:** *Continually searching for ways to optimise operation.* Due to the complex nature and ever changing environment under which DSSs operate in, finding an operational optimum is a challenging task. A couple of approaches have been proposed, introspection [82], and recently a more ad-hoc approach [134] inspired by the self-organising behaviour found in biological systems. The process of introspection is a structured three stage cyclical process: data is collected, analyzed and acted upon. To illustrate, a system samples workload data and upon analysis finds the user to be mostly reading data, the system can then optimise operation by heavily caching on the client side, improving performance for the user and lessening the load on the file server. Several efforts focusing on self-optimisation include GLOMAR [29], HAC [20] and a couple of proposals [85, 84] which apply data mining principles to optimise storage access. GLOMAR is an adaptable consistency mechanism that selects an optimum consistency mechanism based upon the user's connectivity. HAC (Hybrid Adaptive Caching) proposes an adaptable caching mechanism which optimises caching to suit locality and application workload.
3.**Self-healing:** *Being able to recover from component failure.* Large scale distributed storage systems consist of many components and therefore occurrence of failure is to be expected. In an autonomic system, mechanisms to detect and recover from failure are important. For example, DSSs which employ replication to achieve redundancy and better availability need recovery mechanisms when replicas become inconsistent.
4.**Self-protection:** *Be able to protect itself from malicious behaviour or cascading failures.* Systems which operate on the Internet are particularly vulnerable to a wide array of attacks. Self-protection is especially important to these systems. To illustrate, peer-to-peer systems are designed to operate in an untrusted environment and by design adapt well to change be-it malicious or otherwise. Systems which focus on providing anonymity and anti-censorship (Freenet [24] and Free Haven [41]) accommodate for a large array of attacks aimed to disrupt services and propose various methods to protect themselves.

Table 2.2: autonomic computing and distributed storage

discovery, the management and scheduling of resources in a large scale and dynamic environment.

In relation to distributed storage, federation involves understanding the data being stored, its semantics and associated meta-data. The need for managing data has been identified across various scientific disciplines (Ecological [78], High Energy Physics [71], Medicinal [14]). Currently most institutions maintain their own repository of scientific data, making this data available to the wider research community would encourage collaboration. Sharing data across institutions requires middleware to federate heterogeneous storage systems into a single homogeneous interface which may be used to access data. Users need not be concerned about data location, replication and various data formats and can instead focus on what is important, making use of the data. The Data Grid [22] and SRB [7, 106] (Section 2.2.8) are examples of current research being carried out into federating storage services.

2.1.9 Routing and Network Overlays

The evolution of routing has evolved in step with distributed storage architecture. Early DSSs [95, 74, 121] that were based on a client-server architecture, employed a static approach to routing. A client would be configured with the destination address of the server, allowing the client to access storage services in one hop. The server address would seldom change and if so would require the client to be re-configured.

The next phase of evolution in routing was inspired by research into peer-to-peer systems, which itself underwent many stages of development. Early systems like Napster [102] adopted a centralised approach, where peer-to-peer clients were configured with the address of a central peer-to-peer meta-server. This meta-server was responsible for managing a large dynamic routing table which mapped filenames to their stored locations. Clients now required three hops to reach the destined data source: one to query the meta-server for the host address storing the data of interest, another hop for the reply and finally a third hop to the host containing the data. The centralisation introduced by the meta-server proved to be a scalability and reliability bottleneck, inspiring the next generation of peer-to-peer systems.

A method of broadcasting queries [102] was employed by Gnutella to abate centralisation, although this inadvertently flooded the network. Peer-to-Peer clients would broadcast their queries to immediately known peers which in turn would forward the queries to their known list of peers. This cycle of broadcasting flooded the network to the point where 50% of the traffic was attributed to queries [30]. To limit the flooding, a Time To Live (TTL) attribute was attached to queries, this attribute was decremented with every hop. Unfortunately a TTL meant searches would fail to find data even though it was present on the network. The problem of flooding inspired the use of super nodes (FastTrack [39]). Super nodes are responsible for maintaining routing knowledge for a neighbourhood of nodes and serving their queries. The use of super nodes reduced the traffic spent on queries but resulted in a locally centralised architecture.

The next generation of peer-to-peer systems brought routing to the forefront of research. The introduction of Distributed Hash Tables (DHT) spawned much research [105, 154, 135, 118, 107, 31, 90] into network overlays. Routing tables were no longer the property of a centralised meta-server or super nodes, routing tables now belonged to every peer on the network.

Each peer is assigned a hash id, some methods use a random hash, others hash the IP address of the node [154, 118]. Each data entity is referenced by a hash of its payload and upon insertion is routed towards nodes with the most similar hash id. A peer-to-peer network overlay is able to route a peer's storage request within $logN$ hops, where N is the number of nodes in the network. Whilst this may not perform as well as an approach with constant lookup time, network overlays scale well and continue to operate in an unreliable and dynamic environment. A comparison (Table 2.3) of all discussed routing algorithms, suggest that each has a varying capability regarding performance. Variables listed in Table 2.3 are described in detail in [86], which also provides a detailed description and comparison of network overlays.

Continuous research and development into network overlays has seen them evolve to support an increasing number of services. Some of these services include providing stronger consistency [87], better query capability [65, 141], anonymity [54] and censorship resistance [69]. To consolidate the vast array of research, [31] proposes a

System	Model	Hops to Data
AFS, NFS	Client-Server	$O(1)$
Napster	Central Meta-Server	$O(3)$
Gnutella	Broadcasting	$O(TTL)$
Chord	Uni-Dimensional Circular ID space	$O(logN)$
CAN	multi-dimensional space	$O(d.N^{\frac{1}{d}})$
Tapestry	Plaxton-style Global Mesh	$O(log_bN)$
Pastry	Plaxton-style Global Mesh	$O(log_cN)$
Kademlia	X-OR based Look-up Mechanism	$O(log_eN)$

Where:
N: the number of nodes in the network
d: the number of dimensions
b: base of the chosen peer identifier
c: number of bits used for the base of the chosen identifier
e: number of bits in the Node ID

Table 2.3: comparison of routing mechanisms

standard interface for network overlays. The authors hope that standardising will help facilitate further innovation in network overlays and integrate existing peer-to-peer networks. Currently, a user requires a different client to log into every peer-to-peer network, if the standard is embraced, it would serve to integrate various networks, allowing a single client to operate across multiple networks concurrently.

An interesting observation in the evolution of routing is the shift from (1) static centralised routing tables, to (2) static decentralised to (3) dynamic centralised and finally to (4) dynamic decentralised (Figure 2.4). The shift from centralised to decentralised has seen the move from one static server to multiple static servers, replicating storage, providing better redundancy and load balancing. The shift from static to dynamic routing has resulted in storage systems being able to cope with a dynamic environment where each host is capable of providing services. The more recent advance being dynamic decentralised routing tables which has moved the management of routing tables to the *fringes* of the network, giving rise to peer-to-peer network overlays.

	Centralised	**Decentralised**
Static	**1.** Client-Server NFS [121]	**2.** Replicated Servers xFS [4], Coda [124]
Dynamic	**3.** Centralised Peer-to-Peer Napster [102]	**4.** Peer-to-Peer Network Overlay Ivy [97] OceanStore [82]

Table 2.4: routing and architecture taxonomy

2.2 Survey of Distributed Storage Systems

Our survey covers a variety of storage systems, exposing the reader to an array of different problems and solutions. For each surveyed system, we address the underlying operational behaviour, leading into the architecture and algorithms employed in the design and development. Our survey covers systems from the past and present, Table 2.5 lists all the surveyed systems tracing back to those characteristics discussed in the taxonomy.

2.2.1 OceanStore

OceanStore [82] is a globally scalable storage utility, allowing consumers to purchase and utilise a persistent distributed storage service. Providing a storage utility inherently means that data must be highly available, secure, easy to access and support guarantees on Quality of Service (QoS). To allow users to access their data easily from any geographic location, data is cached in geographically distant locations, in effect, travelling with the user and thus giving rise to the term *nomadic data*. OceanStore provides a transparent, easily accessible filesystem interface, hiding any underlying system complexities whilst enabling existing applications to utilise storage services.

Architecture

OceanStore employs a 2-tier based architecture (Figure 2.8), the first is the super-tier, responsible for providing an interface, consistency mechanisms and autonomic operation. It achieves this by maintaining a primary replica amongst an "inner ring" of powerful, well connected servers. The second tier, the archival-tier, is responsible

System	System Function	Architecture	Operating Environment	Consistency	Routing	Interfaces	Scalability
OceanStore	Custom	Locally Centralised Peer-to-Peer	Untrusted	Optimistic	Dynamic DHT (Tapestry)	POSIX and custom	Large (global)
Free Haven	Publish/Share	Pure Peer-to-Peer	Untrusted	N/A (WORM)	Dynamic Broadcast	Custom	Large (global)
Farsite	General purpose Filesystem	Locally Centralised Peer-to-Peer	Partially Trusted	Strong	Dynamic DHT	POSIX	Medium (institution)
Coda	General purpose Filesystem	Locally Centralised	Partially Trusted	Optimistic	Static	POSIX	Medium (institution)
Ivy	General purpose Filesystem	Pure Peer-to-Peer	Trusted	Optimistic	Dynamic DHT (Dhash)	POSIX	Medium (small groups)
Frangipani	Performance	Locally Centralised	Trusted	Strong	Static Petal	POSIX	Medium (small groups)
GFS	Custom	Locally Centralised	Trusted	Optimistic	Static	Incomplete POSIX	Large (institution)
SRB	Federation Middleware	Locally Centralised	Trusted	Strong	Static	Incomplete POSIX	Large (global)
Freeloader	Custom	Locally Centralised	Partially Trusted	N/A (WORM)	Dynamic	Incomplete POSIX	Medium (institution)
PVFS	Performance	Locally Centralised	Trusted	Strong	Static	POSIX, MPI-I/O	Medium (institution)

Table 2.5: distributed storage systems surveyed

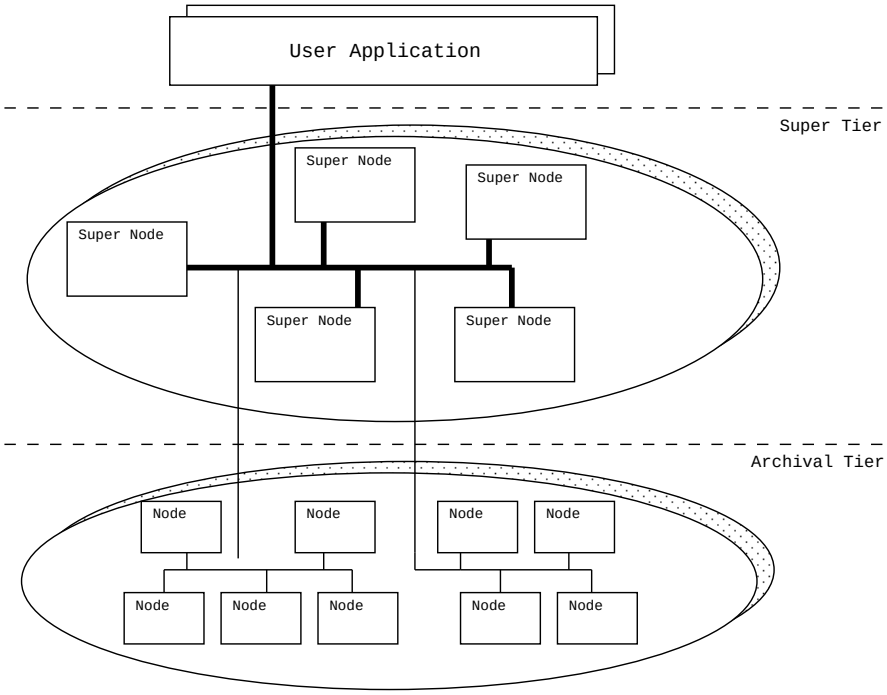

Figure 2.8: OceanStore architecture

for archiving data and providing additional replication by utilising nodes which may not be as well connected or powerful. A hierarchy exists between the tiers, the super-tier constitutes of super nodes, which form a Byzantine agreement [21] enabling the collective to take charge and make decisions. The archival-tier receives data from the super-tier which it stores, providing an archival service. The nodes within an archival-tier need not be well connected or provide high computational speed, as it neither performs high computational tasks or service requests directly made by user applications. The super-tier is a centralised point, as it forms a gateway for users to access their files, but as OceanStore can accommodate multiple cooperating super-tiers, we classify its architecture as locally centralised.

Any requests to modify data are serviced by the super-tier, and hence it is responsible for ensuring data consistency [9]. The super-tier maintains a primary replica which it distributes amongst its nodes. Modifications consist of information regarding the changes made to an object and the resulting state of the object, similar to that of the Bayou System [35]. Once updates are committed to the primary replica, these changes are distributed to the secondary replicas. Before data is distributed to secondary replicas, erasure codes [11] are employed to achieve

redundancy. Erasure codes provide redundancy more efficiently then otherwise possible by replication [148].

The super-tier utilises Tapestry [154], for distributing the primary replica. Tapestry is a peer-to-peer network overlay responsible for providing a simple API capable of servicing data requests and updates, whilst taking care of routing and data distribution to achieve availability across a dynamic environment. Further information on network overlays can be found in Section 2.1.9.

Data objects are stored (read-only), referenced by indirect blocks, in principle very much like a log structured filesystem [114]. These indirect blocks themselves are referenced by a root index. Therefore, when an update is made to a data object, a new pointer is created in the root index, which points to a series of indirect blocks, which finally point to a combination of old unchanged data objects and newly created data objects containing the modifications. This logging mechanism enables every version of the data object to be recreated, enabling the user to recreate past versions of the file, hence the provision of a rollback facility. Unfortunately, providing this feature bears a high cost in space overhead. Indirect blocks are indexed by a cryptographically secure hash of the filename and the owner's public key, whereas data blocks are indexed by a content hash.

Finally, the concept of introspection is introduced as a means of providing autonomic operation. A three-step cycle of Computation, Observation and Optimisation is proposed. Computation is considered as normal operation, which can be recorded and analysed (Observation). Based on these Observations, Optimisations can be put in place.

Implementation

A prototype named Pond [109] has been developed and released as open source under the BSD license and is available for download[1]. SEDA [150] (Staged Event-Driven Architecture) was utilised to provide a means of implementing an event driven architecture. Java was the overall language of choice due to its portability,

[1]OceanStore Homepage: http://oceanstore.cs.berkeley.edu

strong typing and garbage collection. A problem with the unpredictability of the garbage collection was highlighted as an issue, as it was found to pause execution for an unacceptable amount of time posing a performance problem.

Summary

The authors of OceanStore set themselves a very challenging set of requirements covering many areas of research. OceanStore aims to provide a storage utility with a transparent filesystem like interface, providing QoS typical of a LAN whilst operating in a untrusted environment. Providing a storage utility implies the need for accountability and thus a payment system. Providing accountability within a distributed untrusted environment is a challenging task and it would have been interesting to see how that would have been incorporated into the architecture.

The prototype [109] has been tested in a controlled environment and showed promising benchmark results. Pond provides an excellent insight into the challenges of building a system of this calibre. Challenges identified include performance bottlenecks in erasure codes, providing further autonomic operation, increased stability, fault tolerance and security [47].

2.2.2 Free Haven

Free Haven [41] [102] is a distributed storage system which provides a means to publish data anonymously and securely. The aim is to provide individuals with an anonymous communication channel, allowing them to publish and reach out to an audience without the fear of persecution from government bodies or powerful private organisations who would otherwise censor the information. The authors motivation for providing an anonymous communication medium is based on the shortcomings in existing peer-to-peer publication systems, where system operators (Napster [102]) or users themselves (Gnutella [102]) were being persecuted for breach of copyright laws. Performance and availability are secondary with the primary focus being on protecting user identity. Protecting user identity enables individuals to distribute and access material anonymously. Dingldine [40] provides a detailed classification

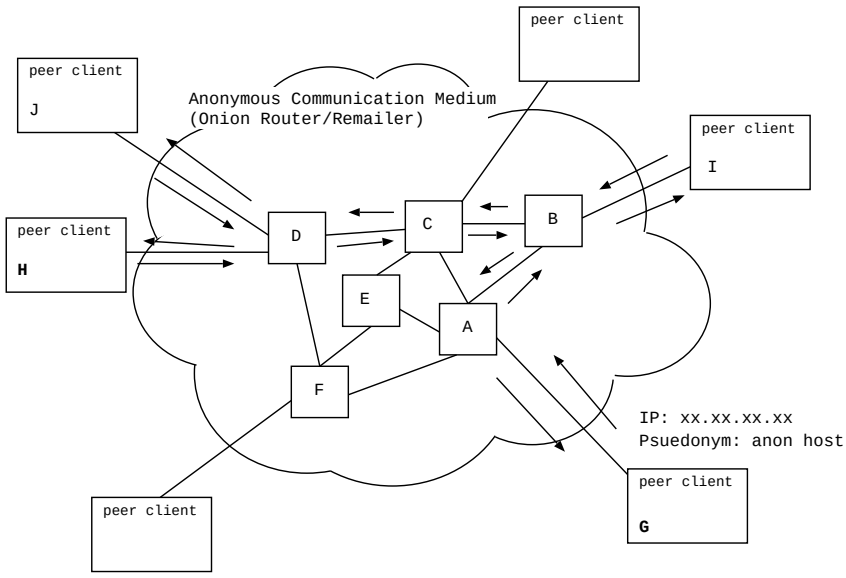

Figure 2.9: Free Haven architecture

of various types of anonymity. Further objectives include (i) persistence: to prevent censorship despite attacks from powerful adversaries, (ii) flexibility: accommodate for a dynamic environment and (iii) accountability: to establish synergy in an otherwise untrusted environment.

Architecture

Free Haven is based upon a pure peer-to-peer design philosophy. With no hierarchy, every node is equal to the next and transactions are carried out in a symmetric and balanced manner. Free Haven utilises a re-mailer network [33], which provides an anonymous communications medium by utilising onion routing (Figure 2.9). Queries are broadcast with the use of onion routing making it difficult for adversaries to trace routes. Each user is assigned a pseudonym to which a reputation is assigned. Servers are only known by their pseudonyms making them difficult to locate. Reputations are assigned to each pseudonym and are tracked automatically. In the rest of this section we shall provide an overall high-level walk-through and discuss reputation and the process of trading.

The primary purpose of the anonymous communication medium is to ensure the messages relayed cannot be traced to the source or destination, protecting user identity. The anonymous communication medium can utilise onion routing or a re-

mailer, both work on a similar set of principles. Nodes communicate by forwarding messages randomly amongst each other using different pseudonyms at each hop making it difficult for adversaries to determine a message's origin or destination. Figure 2.9 shows a peer client G communicating to H along a route that involves nodes A, B, C, D, I and J. Only node A is able to map G's pseudonym to its IP, as once the message is passed beyond A only pseudonyms are used. Even though peer client G may need only to communicate with H, the route taken may visit other peer clients (I and J) along the way, again to make the process of finding a users true identity more difficult.

A reputation mechanism is used to provide an incentive for users to participate in an honest manner. Reputation makes the users accountable, providing a means to punish or even exclude users who misbehave. The process of integrating a reputation mechanism requires careful consideration [42, 88], so as not to compromise the very thing the system was designed to protect, user identity. The amount of data which a server may store is governed by reputation, making it difficult for users to clog the system with garbage. Reputation is calculated based upon the trades a server makes with other servers. A successful trade increases the server's reputation. Trades made amongst servers consist of two equally sized contracts, which are negotiated and (if successful) traded. The size of the contract is based on the file size and the duration for which the file is to be stored. Therefore, the size of contract equates to the file size multiplied by the duration. As such, the larger the file and the longer the period it is to be stored, the more expensive the contract. Servers within the Free Haven environment are continually making trades to: provide a cloak of anonymity for publishers, create a moving target, provide longer lasting shares, and allow servers to join and leave, amongst other reasons.

The process of confirming a trade is made difficult by the fact that it is done in an untrusted environment. Detecting malicious behaviour, where servers may falsely deny they received a trade or present false information about another server to reduce its reputation. To address these problems, a buddy system is introduced which involves each server having a shadow to look over and certify trades. When a negotiation of a contract is finalised, each server sends a receipt acknowledging the

trade. This receipt is then sent from each server to the other and to their respective buddies. Each buddy will receive the same receipt twice. Once from the server which created the trade and once from the accepting server. This enables the buddies to oversee the contract and detect any malicious behaviour.

Implementation

Free Haven has not been released, the website details problems and areas which need to be addressed and as such development is in a state of hibernation[2]. The problems discussed involve:

1. *Reputation:* Flaws have been identified in the current reputation system with a need to incorporate verifiable transactions.

2. *Protocol:* The underlying protocol is based on broadcasting messages, this was found to be too inefficient.

3. *Anonymous Communication:* At the moment there is no anonymous communications medium. An enhanced version of the onion routing protocol is proposed [43], detailing how anonymity could be integrated at the TCP level rather than at the message level. Although weaker anonymity is traded against lower latency in this situation.

Releases of both the the anonymous re-mailer Mixminion [33] and Tor [43] can be found on the Free Haven website.

Summary

Free Haven aims to operate in a globally untrusted environment providing the user with the ability to anonymously publish data. Free Haven sacrifices efficiency and convenience in its pursuit of anonymity, persistence, flexibility and accountability. The persistence of data published is based on duration as apposed to popularity (as in many other publishing systems), this is an important unique feature as it prevents

[2]Free Haven Homepage: http://www.freehaven.net/

popular files from *pushing out* other files and as such cannot be used by adversaries to censor information.

As Free Haven aims to resist censorship and provide strong persistence, even under attacks from strong opponents, its design was based on detailed consideration [40] of possible attacks. The documented attacks are applicable to any system operating in an untrusted environment. The concepts applied by Free Haven to achieve anonymity could be applied by other systems aiming to protect user privacy.

2.2.3 Farsite

The goal of Farsite [2] is to provide a secure, scalable file system by utilising unused storage from user workstations, whilst operating within the boundaries of an institution. Farsite aims to provide a transparent, easy to use file system interface, hiding its underlying complexities. From the administrators perspective, it aims to simplify effort required to manage the system. Tasks such as backing up are made redundant through replication, available storage space is proportionate to the free space on user machines. This autonomic behaviour aims to reduce the cost of ownership by simplifying the administration and better utilising existing hardware. If a need for further storage is required, the option of adding dedicated workstations to the network can be achieved without introducing down time. Due to its ability to utilise existing infrastructure, Farsite can be seen as a cheaper solution to a SAN, but only if a trade-off in performance is acceptable.

Architecture

The architecture is based on the following three concepts: client, directory group and a file host (Figure 2.10). A node may adopt any, or all of these roles. The client is responsible for providing a filesystem like interface. Nodes which participate in a directory group do so in a Byzantine agreement, these nodes are responsible for establishing trust, enforcing consistency, storing meta-data and monitoring operational behaviour. They can also, as required, execute choirs and exhibit a degree of autonomic operation. The file host role consists of providing storage space

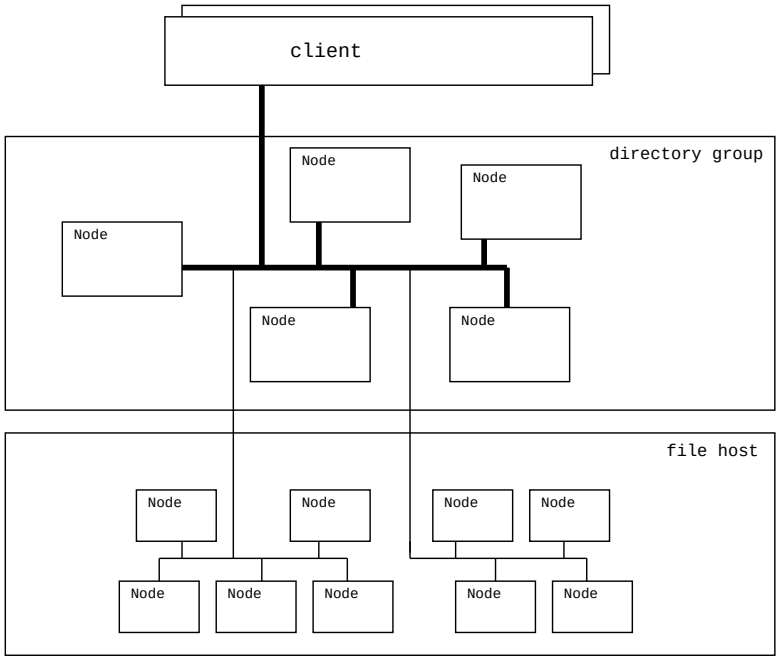

Figure 2.10: Farsite architecture

for file data. We shall now discuss each role in greater detail.

The client provides an interface which emulates the behavior of a traditional local filesystem, providing users with the ability to access the system in a transparent, easy to use manner. The directory group begins as a set of nodes assigned the root namespace for which they have to service client requests. As the namespace grows, a part of the namespace is delegated to another directory group. Each group establishes trust and redundancy via the Byzantine protocol. Every node in this group maintains a replica of the meta-data. The directory group behaves like a gateway for client requests, ensuring consistency by utilising leases. Autonomic behavior extends to managing replication, by relocating replicas to maintain file availability. File availability is based on the availability of replicas and therefore files which have a higher availability than the mean availability have their replicas swapped with replicas which have lower availability, this establishes a uniform level of availability across all files.

The meta-data stored by the directory group includes certificates, lease information, directory structure, Access Control Lists (ACL) and a routing table, consisting of filenames, content hash and file location. There are three main types

of certificates, a namespace certificate which associates the root of a file-system namespace with the directory group, a user certificate which associates the user with his public key, to provide a means to authorise a user against an ACL and a machine certificate which is similar to the user certificate except it is used to authorise and identify the machine as a unique resource. Certificates are signed by trusted authorities, which are used to establish a chain of trust. A user's private key is encrypted with a symmetric key derived from the user's password.

Farsite utilises leases to ensure consistency. The granularity of leases is variable, in that they may cover anything from a single file to a directory tree. There are four main types of leases, content leases, name leases, mode leases and access leases. Content leases govern what access modes are allowed. There are two types of content leases, read-write which permits a client to perform both read and write operations and a read-only lease that guarantees the client that data read is not stale. Name leases provide clients with control over a filenames in a directory. Mode leases are application level leases, enabling applications to have exclusive read, write or delete modes. Access leases are used to support Microsoft Windows deletion semantics, which state that a file can be marked to be deleted, but can only be deleted after all open handles are released. A file that is marked for deletion cannot accept new file handles, but applications which already hold a file handle have the capability of resetting the delete flag. To support this there are three types of access leases; public, protected and private. A public lease being the least restrictive of the three, indicates the lease holder has the file open. A protected lease is the same as the public lease with the extra condition that any lease request made by clients must first contact the lease holder. Finally the private lease is the same as the protected lease but with a further condition that any access lease request by a client will be refused.

Implementation

Unfortunately Farsite is closed source and because of this, limited information is available[3]. The authors break down the code into two main components, user and kernel level, both developed in C. User level component is responsible for the backend jobs, including managing cache, fetching files, replication, validation of data, lease management and upholding the Byzantine protocol. Kernel level component is mainly responsible for providing a filesystem like interface for the user. Whilst Farsite has implemented some of its proposed algorithms, others remain to be completed, including those related to scalability and crash recovery.

Summary

Farsite aims to operate in a controlled environment, within an institution. The controlled nature of this environment means that nodes are assumed to be interconnected by a high bandwidth, low latency network and whilst some level of malicious behaviour is expected, on the whole, most machines are assumed, to be available and functioning correctly. As a level of trust is assumed we classify the operating environment as partially trusted. Farsite bases its workload model on typical desktop machine operating in a academic or corporate environment and thus assumes files are not being updated or read by many concurrent users. Farsite maintains a database of content hashes of every file and utilises it to detect duplicate files and increase its storage efficiency. On the whole Farsite aims to provide distributed storage utilising existing infrastructure within an institution whilst minimising administration costs, through autonomic operation.

2.2.4 Coda

Coda [124, 123, 81] provides a filesystem like interface to storage that is distributed within an institution. Coda clients continue to function even in the face of network outages, as a local copy of the user's files is stored on their workstation. As well as providing better resilience to network outages, having a local copy increases

[3]Farsite Homepage: http://research.microsoft.com/Farsite/

performance and proves to be particularly useful to the ever growing group of mobile users taking advantage of laptops. Coda was designed to take advantage of Conventional Off The Shelf (COTS) hardware, proving to be a cost competitive solution compared with expensive hardware required by traditional fileservers or SANs. Upgrades simply require the addition of another server, without affecting the operation of existing servers, therefore eliminating unavailability due to upgrades. Coda has been designed to operate within an institution and its servers are assumed to be connected by a high bandwidth, low latency network in what we deem to be a partially trusted environment.

Architecture

Coda is divided into two main components (Figure 2.11), the server (Vice) and the client (Venus). Many Vice servers can be configured to host the same Coda filesystem in effect replicating the filesystem. Each Vice server that hosts the filesystem is part of a Volume Storage Group (VSG). Referring to Figure 2.11, we can see that Vice Servers A, B and C form a VSG for volume A, whilst only Vice Servers B and C form a VSG for volume B. The Venus client software enables the user to mount the Coda volume, providing a transparent filesystem interface. Venus has knowledge of all available Vice servers and broadcasts its requests to them. Venus caches frequently accessed files allowing users to operate on cached files even when disconnected from Vice servers.

The architecture of Coda is heavily oriented around the client. The client is left with the majority of the responsibilities, reducing the burden and complexity of the Vice server. Therefore, the client is left with the responsibility for detecting inconsistencies and broadcasting changes to all Coda servers. This itself could prove to be a bottleneck as the system scales up.

Clients have two modes of operations, a connected mode when the Client has connectivity to the Server and a disconnected mode when the client loses connectivity to the server. Disconnected mode enables the user to continue operation even whilst losing connectivity with the network. Coda is able to provide this mode by caching files locally on the user's machine. Whilst caching was initially seen as a means to

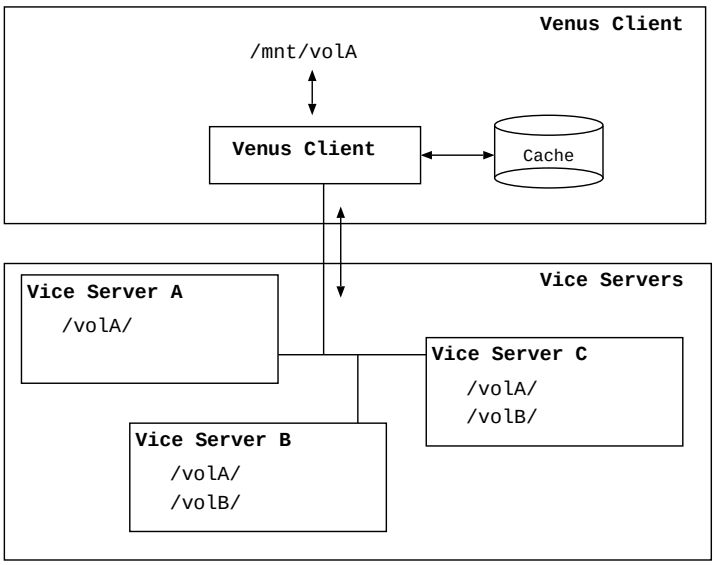

Figure 2.11: Coda architecture

improve performance, it has the added advantage of increasing availability. Files are cached locally based upon the Least Recently Used (LRU) algorithm, much like traditional caching algorithm. Allowing client side caching and disconnected operation raises issues relating to consistency.

There are two possible scenarios leading to data inconsistency in the Coda environment. The first is in the event that a client enters disconnected operation, the second being when a Coda server loses connectivity with other Coda servers. When a client switches to disconnected operation the user is still able to make changes as if they were still connected, completely oblivious to the fact they have lost connectivity. Whilst the user makes changes a log is kept of all the changes they make to their files. Upon reconnection an attempt to merge their changes with the Coda server is attempted by replaying the log of their changes. If the merge fails and a conflict is detected, manual intervention is required to resolve the conflict.

Coda's approach to consistency is optimistic as it allows data replicas to become inconsistent. To illustrate, disconnected users are permitted to make changes and hence their local replica becomes inconsistent with the server's, only when the user reconnects are all replicas returned to a consistent state. The choice to use an optimistic approach was based on analysing a users' workload profile [81] and observing that it was an unlikely occurrence for them to make modifications where

a conflict would arise. With this in mind, the advantages to be gained by optimistic concurrency control far outweigh the disadvantages.

When a Coda server loses connectivity with other servers, the responsibility of detecting inconsistencies is left with the client. When a client requests a file, it first requests the file version from each of the Coda servers. If it detects a discrepancy in the version numbers, it notifies the Coda server with the latest version of the file. It is only then that changes are replicated amongst the Coda servers.

Implementation

Coda was written in C and consists of two main components, the Vice server and the Venus client (Figure 2.12). Venus consists of two main modules, the Coda FS kernel module and the cache manager. The Coda FS kernel module is written to interface the Linux VFS (virtual file system) enabling it to behave like any other filesystem. When a client program accesses data on a Coda mount point, VFS receives these I/O requests and routes them to the Coda FS kernel module. The Coda FS kernel module than forwards these requests to cache manager, which, based on connectivity and cache status, can choose to service these requests by either logging them to local store or contacting the Vice servers. Vice consists of one main component which provides an RPC interface for Venus to utilise in the event of cache misses or meta-data requests.

Coda is an open source effort and is available for download[4]. Whilst Coda itself is written in C, the distribution is accompanied by a host of utilities written in shell and Perl for recovery and conflict resolution. Current development efforts include: making Coda available to a wider community by porting it to various popular platforms, reliability, robustness, setting up a mailing group and extending the available documentation.

[4]Coda Homepage: http://www.coda.cs.cmu.edu/

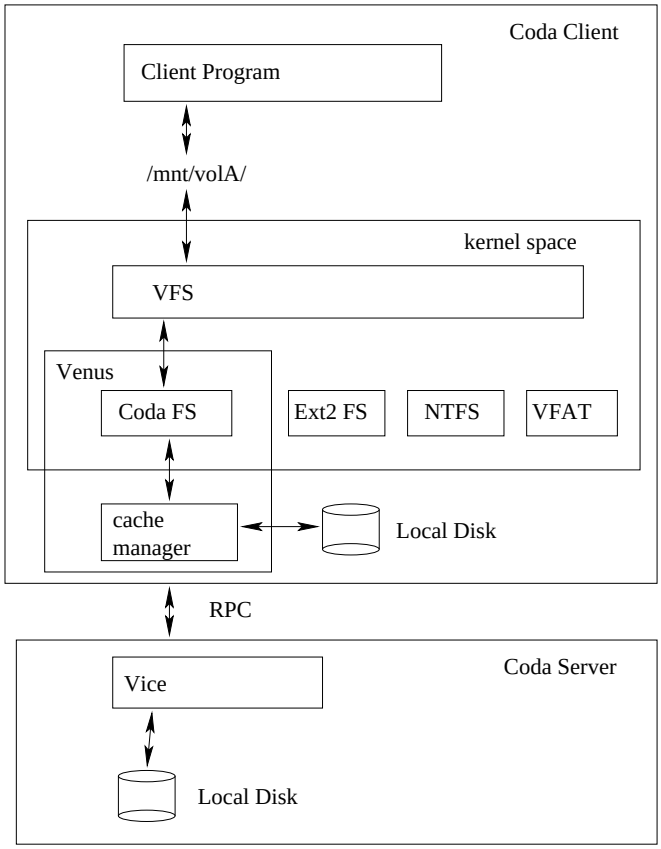

Figure 2.12: Coda implementation architecture

Summary

Coda aims to provide all the benefits associated with conventional file servers whilst utilising a decentralised architecture. Coda is resilient to network outages by employing an optimistic approach to consistency, which allows clients to operate on locally cached data whilst disconnected from the server. Utilising an optimistic consistency model is a key component in providing maximum data availability, although this creates the possibility for consistency conflicts to arise. Knowledge gained from the usage of Coda [81] has shown that the occurrence of conflicts are unlikely and therefore the advantages gained by utilising an optimistic consistency model outweigh the disadvantages. Coda's ability to provide disconnected operation is a key unique feature, which will grow in popularity with mobile computing.

2.2.5 Ivy

Ivy [97] employs a peer-to-peer architecture to provide a distributed storage service with a filesystem like interface. Unlike many existing peer-to-peer storage systems (Gnutella [102], Napster [102]) which focus on publishing or at best only supporting the owner of the file to make modifications, Ivy supports read-write capability and an interface which is indifferent to any other mounted filesystem. Ivy is suited to small cooperative groups of geographically distant users. Due to its restrictive user policy, a user is are able to choose which other users to trust. In the event a trusted user node is compromised and changes made are malicious, a rollback mechanism is provided to undo any unwanted changes. Ivy is designed to be utilised by small groups of cooperative users in an otherwise untrusted environment.

Architecture

Ivy's architecture has no hierarchy, with every node being identical and capable of operating as both a client and server. Due to its symmetrical nature, the architecture is considered pure peer-to-peer. Each node consists of two main components Chord/Dhash and the Ivy server(Figure 2.13). Chord/Dhash is used for providing a reliable peer-to-peer distributed storage mechanism. The Ivy server interfaces to Dhash, to send and receive data from peer nodes, and to the NFS loop-back to provide a filesystem interface.

Ivy uses a log based structure whereby every user has their own log and view of the filesystem. Logs contain user data and the changes made to the filesystem. These logs are stored in a distributed fashion utilising Chord/DHash [135], a peer-to-peer network overlay (Section 2.1.9), used for its ability to reliably store and retrieve blocks of data across a network of computers.

The log contains a linked list data structure, where every record represents one NFS operation. Log records are immutable and kept indefinitely enabling users to roll back any unwanted changes, much like a log structured filesystem [114]. Whilst Ivy supports file permission attributes, all users are able to read any log in the Ivy system. It is advised that if a user wishes to restrict access to their files they use

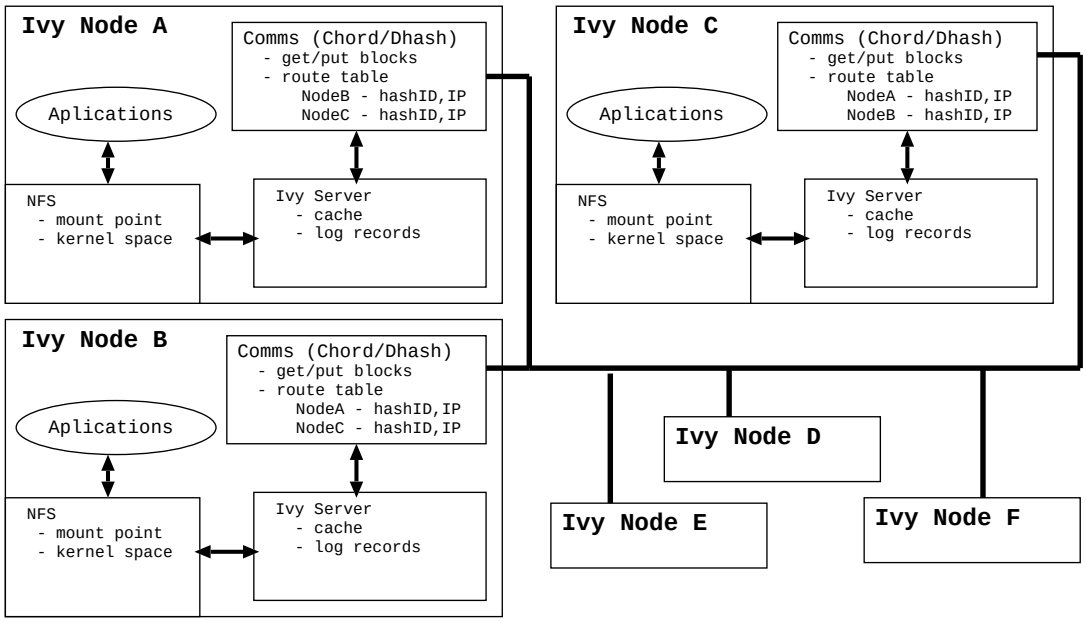

Figure 2.13: Ivy architecture

encryption. Log records store minimal information to minimise the possibility of concurrent updates and consistency issues.

To create a filesystem within Ivy, a group of users agree upon which set of logs will be trusted and therefore used to generate the filesystem. For every log deemed to be part of the filesystem, an entry pointing to its log head is created in the view array. The view array is the root index and is traversed by all the users to generate a snapshot of the filesystem. A filesystem may comprise of multiple logs which in turn can be used to record modifications concurrently. As Ivy supports concurrent writes, consistency conflicts can occur.

Ivy aims to provide close-to-open consistency and as such modifications completed by users are immediately visible to operations which other participants may initiate. This feature cannot be upheld when nodes in the Ivy filesystem lose connectivity or become partitioned. To achieve close-to-open consistency, every Ivy server that is performing a modification waits until Dhash has acknowledged the receipt of new log records before announcing completion. For every NFS operation, Ivy requests Dhash for the latest view array. Modifications which result in consistency conflicts require the use of the *lc* command, which detects conflicts by traversing the logs, looking for entries with concurrent version vectors which affect

the same file or directory entry. Users are expected to resolve these conflicts by analysing the differences and merging the changes.

Whilst an optimistic approach to consistency is used with respect to file modifications, a more strict strategy (utilising locking) is in place for file creation. Ivy aims to support exclusive creation, its reason for doing so extends to applications which rely upon these semantics to implement their own locking. Ivy can only guarantee exclusive creation when the network is fully available. As each user has to fetch every other user's log, performance degrades as the number of users increase. Consequently, Ivy's scalability is limited and hence the system is only suited to small groups of users.

Implementation

Ivy is distributed as open source under the GPL agreement and is available for download[5]. Source code is written using a combination of C and C++. The SFS tool kit is utilised for event-driven programming. Performance benchmarks conducted in a dedicated controlled environment and with replication switched off in Chord/DHash, showed promising results where Ivy was only a factor of 2 to 3 times slower than NFS.

Summary

Ivy uniquely provides a distributed storage service with a filesystem like interface, whilst employing a pure peer-to-peer architecture. Every user stores a log of their modifications and at a specified time interval generates a snapshot, a process which requires them to retrieve logs from all participating users. Whilst the transfer of logs from every user may prove to be a performance bottleneck, users have the ability to make changes to the filesystem without concern to the state of another participant's logs. Ivy logs and stores every change a user makes which enables users to rollback any unwanted changes, although this comes at a high cost in storage overhead. Ivy utilises an optimistic approach to consistency allowing users to make concurrent

[5]Ivy Homepage: http://www.pdos.lcs.mit.edu/ivy/

changes to the same piece of data, providing users with maximum flexibility whilst avoiding locking issues. Although, like any other systems which adopts an optimistic approach to consistency, the system can reach an inconsistent state requiring user intervention to resolve. Overall, Ivy can be seen as an extension of CFS [32], which like Ivy utilises Chord/DHash for distributing its storage but only supports a limited write-once/read-many interface.

2.2.6 Frangipani

Frangipani [140] is best utilised by a cooperative group of users with a requirement for high performance distributed storage. It offers users excellent performance as it stripes data between servers, increasing performance along with the number of active servers. Frangipani can also be configured to replicate and thus offer redundancy and resilience to failures. It provides a filesystem like interface that is completely transparent to users and applications. Frangipani is designed to operate and scale within an institution and thus machines are assumed to be interconnected by a secure, high bandwidth network under a common administrative domain. The operating environment by nature mirrors a cluster and can be considered a trusted environment. Frangipani was designed with the goal of minimising administration costs. Administration is kept simple even as more components are added. Upgrades simply consist of registering new machines to the network without disrupting operation.

Architecture

Frangipani consists of the following main components: Petal Server, Distributed Locking Service and the Frangipani File Server Module (Figure 2.14). The Petal Server [139] is responsible for providing a common virtual disk interface to storage that is distributed in nature. As Petal Server nodes are added, the virtual disk scales in throughput and capacity. The Petal device driver mimics the behaviour of a local disk, hiding its distributed nature.

The Distributed Locking Service is responsible for enforcing consistency, thus

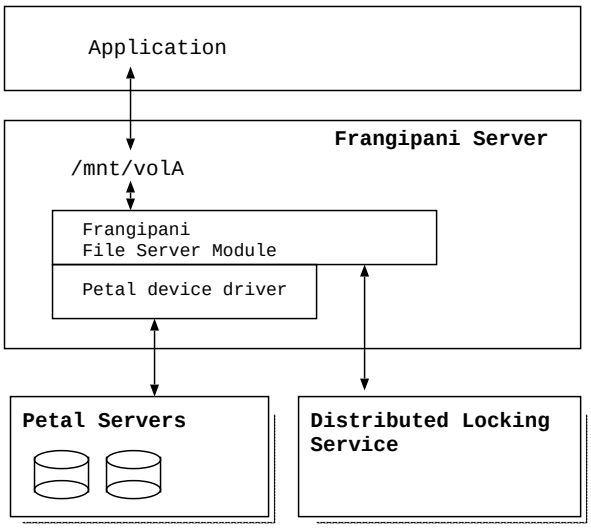

Figure 2.14: Frangipani architecture

changes made to the same block of data by multiple Frangipani servers are serialised ensuring data is always kept in a consistent state. The locking service is an independent component of the system, it may reside with Petal or Frangipani Servers or even on an independent machine. It was designed to be distributed, enabling the service to be instantiated across multiple nodes with the aim of introducing redundancy and load balancing.

The locking service employs a multiple reader, single writer locking philosophy. It employs a file locking granularity where files, directories and symbolic links are lockable entities. When there is a lock conflict, the locking service sends requests to the holders of the conflicting locks to either release or downgrade. The are two main types of locks, a read lock and a write lock. A read lock allows a server to read the data associated with the lock and cache it locally. If it is asked to release its lock, it must invalidate its cache. A write lock permits the server to read and write to the associated data. If it is asked to downgrade, the server must write any cached modifications and downgrade to a read lock. If it is asked to release the lock, it must also invalidate its cache.

The third component is the Frangipani File Server Module, which interfaces with the kernel and the Petal device driver to provide a filesystem like interface. Frangipani File Server communicates with the Distributed Locking Service to acquire locks and ensure consistency, and with Petal Servers for block-level storage

capability. Frangipani File Server communicates with Petal Servers via the Petal device driver module which is responsible for routing data requests to the correct Petal Server. It is the responsibility of Frangipani File Server Module to abstract the block-level storage provided by the Petal device driver and present a file level interface to the kernel, which in turn provides a filesystem interface.

Frangipani utilises write-ahead redo logging of meta-data to aid in failure recovery. The logged data is written into a special area of space allocated within Petal Server. When the failure of a Frangipani File Server is detected, any redo logs written to a Petal Server are used by the recovery daemon to perform updates and upon completion releases locks owned by the failed server.

Implementation

Frangipani was implemented on top of the Petal system, employing Petal's low-level distributed storage services. Frangipani was developed on a DIGITAL Unix 4 environment. Through careful design considerations, involving a clean interface between Petal Server and Frangipani File Server, the authors were able to build the system within a few months. Unfortunately, because of Frangipani's close integration to the kernel, its implementation is tied to the platform, making it unportable to other operating systems. The product has no active web page and seems that its has no active developer/user base. Frangipani is closed source and unfortunately in an archived state.

Summary

Frangipani provides a distributed filesystem that is scalable in both size and performance. It is designed to be utilised within the bounds of an institution where servers are assumed to be connected by a secure high bandwidth network. Performance tests carried out by the authors have shown that Frangipani is a very capable system. A benchmark on read performance showed Frangipani was able to provide a near linear performance increase with respect to the number of Petal Servers. The only limiting factor was the underlying network, with benchmark results tapering off as they approached the limit imposed by network capacity.

An interesting experiment was conducted, showing the effects of locking contention on performance. The experiment consisted of a server writing a file while other servers read the file. The frequent lock contention resulted in a dramatic performance drop, in the factors of 15 to 20. In summary, the impressive benchmark results demonstrate that Frangipani is a capable high performance distributed storage system, whilst being resilient to component failure.

2.2.7 GFS

The Google File System [57] is a distributed storage solution which scales in performance and capacity whilst being resilient to hardware failures. GFS is successfully being utilised by Google to meet their vast storage requirements. It has proven to scale to hundreds of terabytes of storage, utilising thousands of nodes, whilst meeting requests from hundreds of clients. GFS design was primarily influenced by application workload. In brief, GFS is tailored to a workload that consists of handling large files (> 1GB) where modifications are mainly appends, possibly performed by many applications. With this workload in mind, the authors propose interesting unique algorithms. Existing applications may need to be customised to work with GFS as the custom interface provided does not fully comply to POSIX file I/O. Whilst GFS has proven to be scalable, its intended use is within the bounds of an institution and in a Trusted Environment.

Architecture

In the process of designing GFS, the authors focused on a selection of requirements and constraints. GFS was designed to utilise Commodity Off The Shelf (COTS) hardware. COTS hardware has the advantage of being inexpensive, although failure is common and therefore GFS must accommodate for this. Common file size will be in the order of Gigabytes. Workload profile, whether reading or writing, is almost always handled in a sequential streaming manner, as apposed to random. Reads consist of either large streaming reads (MB+) or small random reads. Writes mainly consist of appending data, with particular attention made to supporting multiple

clients writing records to the same file.

Bearing all these constraints and requirements in mind, GFS proposes an interesting solution. Replication is used to accommodate for node failures. As most of the workload is based upon streaming, caching is non-existent, this in turn simplifies the consistency, allowing a more "relaxed model". A special atomic append operation is proposed to support multiple concurrent clients appending without the need to provide synchronisation mechanisms. Having described the core concepts behind GFS, we shall now discuss the architecture.

GFS has three main components (Figure 2.15), a Master Server, Chunk Servers and a Client Module. For an application to utilise the GFS, the Client Module needs to be linked in at compile time. This allows the application to communicate with the Master Server and respective Chunk Servers for its storage needs. A Master Server is responsible for maintaining meta-data. Meta-data includes namespace, access control information, mapping information used to establish links between filenames, chunks (which make up files contents) and their respective Chunk Server locations. The Master Server plays an important role in providing autonomic management of the storage the Chunk Servers provide. It monitors the state of each Chunk Server and in the event of failure, maintains a level of replication by using remaining available replicas to replicate any chunks that have have been lost in the failure. The Chunk Servers are responsible for servicing data retrieval and storage requests from the Client Module and the Master Server.

Having a single Master Server introduces a Single Point of Failure (SPF) and consequently a performance and reliability hot-spot. In response to these challenges, the Master Server replicates its meta-data across other servers, providing redundancy and a means to recover in the event of failure. To avoid the Master Server becoming a performance hot-spot, the Client Module interaction with the Master Server is kept to a minimum. Upon receiving the Chunk Server location from the Master Server, the Client Module fetches the file data directly from the corresponding Chunk Server. The choice of using a large chunk size of 64MB also reduces the frequency with which the Master Server needs to be contacted.

A large chunk size also has the following advantages: it is particularly suited to a

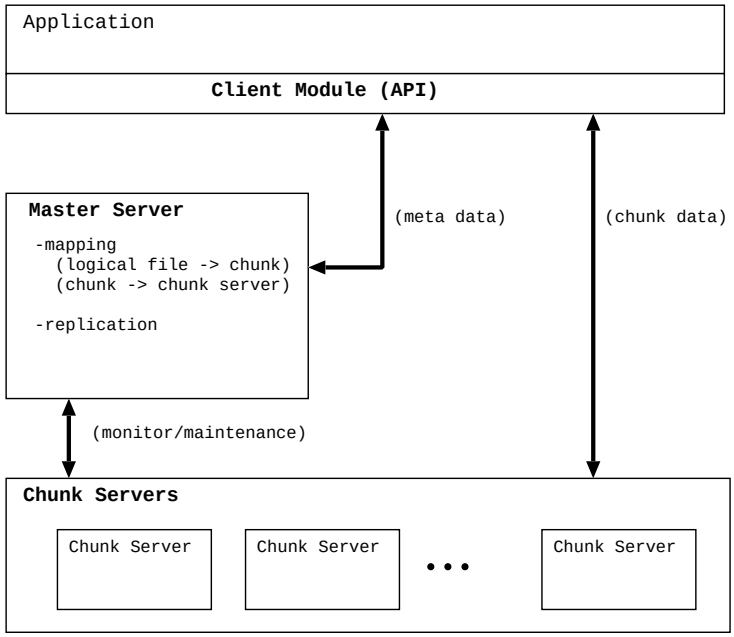

Figure 2.15: GFS architecture

workload consisting of large streaming reads or appends such as GFS, lower network overhead as it allows the Client Module to sustain an established TCP connection with a Chunk Server for a longer period. A disadvantage normally associated with a large chunk size is the wasted space, which GFS avoids by storing chunks as files on a Linux filesystem.

GFS follows an optimistic consistency model, which suites their application requirements well and allows for a simple solution whilst enabling multiple concurrent writers to append to a particular file. This feature is particularly suited to storage requirements of distributed applications, enabling them to append their results in parallel to a single file.

GFS supports two types of file modifications, writes and record appends. Writes consist of data being written at a specified offset. *"A record append causes data to be appended atomically at least once even in the presence of concurrent mutations, but at an offset of GFS's choosing."* Adopting an optimistic approach to consistency (as apposed to implementing distributed locking) introduces the possibility that not all replicas are byte-wise identical, allowing for duplicate records or records that may need to be padded. Therefore, the client is left with the responsibility of handling padded records or duplicate records. The authors acknowledge that consistency and

concurrency issues do exist, but that their approach has served them well.

Implementation

Unfortunately, due to the commercial nature of GFS the source code has not been released and limited information is available. The authors discuss the Client Module utilises RPC for data requests. A discussion into the challenges which they have encountered whilst interfacing to the Linux kernel is also documented. This suggests that a large portion of code, if not all, was written in C.

Summary

GFS was designed to suit a particular application workload, rather than focusing on building a POSIX-compliant filesystem. GFS is tailored to the following workload: handling large files, supporting mostly large streaming reads/writes and supporting multiple concurrent appends. This is reflected in the subsequent design decisions, large chunk size, no requirement for caching (due to streaming nature) and a relaxed consistency model. GFS maintains replication allowing it to continue operation even in the event of failure. The choice of using a centralised approach simplified the design. A single Master Server approach meant that it was fully aware of the state of its Chunk Servers and allowed it to make sophisticated chunk placement and replication choices. Benchmarks have shown GFS to scale well providing impressive aggregate throughput for both read and write operations. GFS is a commercial product successfully being used to meet the storage requirements within Google.

2.2.8 SRB

Data can be stored under many types of platforms in many different formats. Federating this heterogeneous environment is the primary job of the Storage Resource Broker (SRB) [7, 106]. The SRB provides a uniform interface for applications to access data stored in a heterogeneous environment. SRB aims to simplify the operating environment under which scientific applications access their data. Applications accessing data via the SRB need not concern themselves with

locations or data formats, instead they are able to access data with high level ad-hoc queries. Whilst providing a uniform interface, the SRB also enables applications to access data across a wide area network, increasing data availability. The SRB was designed and developed to provide a consistent and transparent means for scientific applications to access scientific data stored across a variety of resources (filesystems, databases and archival systems).

Architecture

The SRB architecture consists of the following main components; the SRB server, Meta-data Catalog (MCAT) and Physical Storage Resources (PSRs). The SRB server is middleware which sits between the PSRs and the applications which access it (Figure 2.16). MCAT manages meta-data on stored data collections, PSRs and an Access Control List (ACL). PSRs refer to the Physical Storage Resource itself, which could be a database, a filesystem or any other type of storage resource for which a driver has been developed. Applications read and write data via the SRB server, issuing requests which conform to the SRB server API. Data stored via the SRB needs to be accompanied by an description which is stored in MCAT. The SRB server receives requests from applications, consults the MCAT to map the request to the correct PSR, retrieves the data from the PSR and finally forwards the result back to the application. SRB servers have a federation mode of operation where one SRB server behaves as a client of another SRB server. This allows applications to retrieve data from PSRs that may not necessarily be under the control of the SRB server they communicate with.

Now that we have a high level understanding of how the major components of SRB work together, we shall provide more details about security, MCAT and the data structures used to manage stored data. Security is broken down into two main areas, authentication and encryption between the application and the SRB server and amongst the SRB servers. The SRB server supports password-based authentication with data encryption based on SEA [129], which employs public and private keys mechanisms and a symmetric key encryption algorithm (RC5). When SRB servers operate in federated mode, the communication between them is also

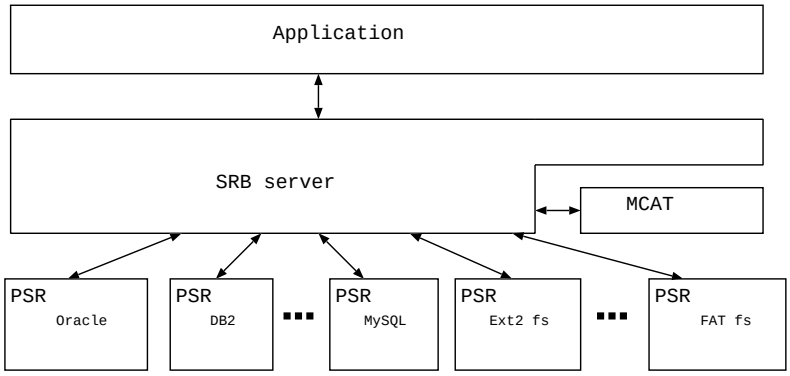

Figure 2.16: SRB architecture

encrypted using the same mechanisms. During authentication the SRB server queries MCAT for authentication details. Data access is controlled by a ticketing scheme whereby users with appropriate access privileges may issue tickets to access objects to other users. These tickets may expire based on duration or the number of times they have been used to access data.

MCAT organises data in a collection hierarchy. The hierarchy is governed by the following axioms: A collection contains zero or more sub-collections or data items. A sub-collection may contain zero or more data items or other sub-collections. A data item is a file or a binary object. This hierarchy scheme extends to data access control. Users, be-it registered or unregistered, are issued with a ticket for every collection they wish to access. This ticket will grant them access to the collection and the subsequent data objects contained within the hierarchy of that collection. PSRs are also organised in a hierarchical manner, where one or more PSRs can belong to a single Logical Storage Resource (LSR). PSRs which belong to the same LSR may be heterogeneous in nature, and therefore the LSR is responsible for providing uniform access to a heterogeneous set of PSRs. Data written to a LSR is replicated across all PSRs and can be read from any PSR as its final representation is identical.

As data is replicated amongst PSRs, there is a possibility for inconsistencies to arise when a PSR fails on a write. SRB handles this scenario by setting the "inconsistent" flag for that replica, preventing any application from accessing dirty data. Replicas which become inconsistent can re-synchronise by issuing a replicate command, which duplicates data from an up to date replica.

When a client connects to an SRB server, it sends a connect request. Upon receiving a connect request, the SRB server will authenticate the client and fork off an SRB agent. The SRB agent will then handle all subsequent communication with the client. SRB allows different SRB servers to communicate between each other, allowing the federation of data across different SRB servers. The SRB agent will query MCAT to map high level data requests to their physical stored locations and if the data request can be serviced by local PSRs the SRB agent will initiate contact with the PSR which is known to have the data.

Implementation

SRB binaries and source code are available for download[6]. Downloading the software requires registration, upon which a public key can be used to decrypt and install SRB. SRB is currently being used across the United States, a major installation being the BIRN Data Grid, hosting 27.8 TB of data across 16 sites. SRB has been developed using a combination of C and Java, providing many modules and portals which support a multitude of platforms, including the web.

Summary

SRB was built to provide a uniform homogeneous storage interface across multiple administrative domains which contain heterogeneous storage solutions and data formats. The homogeneous interface provided by SRB aims to simplify data storage and retrieval for scientific applications which have to deal with many data-sets. This simplification removes the need for scientists to individually implement modules to access data in different formats or platforms. The authors of SRB have identified a possible centralisation bottleneck associated with the MCAT server and wish to do a performance impact study with a large number of concurrent users.

[6]SRB Homepage: http://www.sdsc.edu/srb/

2.2.9 Freeloader

Scientific experiments have the potential to generate large data-sets, beyond the storage capability of end-user workstations, typically requiring a temporary storing hold as scientists analyse the results. Freeloader [143] aims to provide an inexpensive way to meet these storage requirements whilst providing good performance. Freeloader is able to provide inexpensive, mass-storage by aggregating scavenged storage from existing workstations and through the use of striping, is able to aggregate network bandwidth providing an inexpensive but fast alternative to storage offered by a file server. Freeloader is intended to operate within a partially trusted environment and scale well within the bounds of an institution.

Architecture

Freeloader was designed with the following assumptions in mind: (i) usage pattern is expected to follow a write-once/read-many profile, (ii) scientists will have a primary copy of their data stored in another repository, (iii) data stored is temporary (days-weeks) in nature, before new data is generated. Freeloader aims to fulfill these assumptions rather than being a general purpose filesystem. Data is stored in 1MB chunks called *Morsels*, this size was found to be ideal for GB-TB data-sets.

Freeloader consists of three main components (Figure 2.17); Freeloader Client, Freeloader Manager and Benefactor. The Freeloader Client is responsible for servicing user storage requests, in doing so communicates with the Freeloader Manager and respective Benefactors. A Benefactor is a host which donates its available storage, whilst servicing Freeloader Clients' storage requests and meta-data requests from the Freeloader Manager. The Freeloader Manager component is responsible for maintaining system meta-data whilst overseeing the overall operation of the system. The overall architecture of Freeloader shares many similarities to GFS [57] and PVFS [16], even though each system has distinct operational objectives and algorithms. We now discuss each of the main components in greater detail.

The Freeloader Client is responsible for servicing application storage requests by translating incoming function calls to requests, which are then routed to the

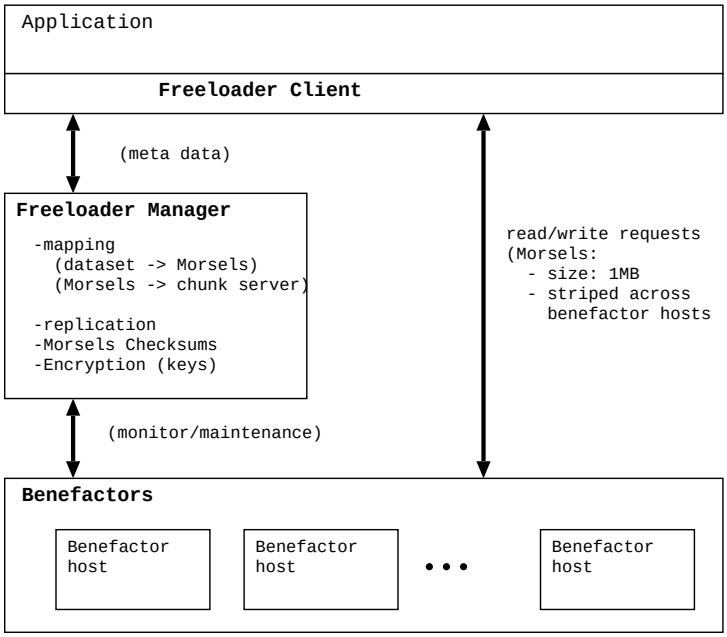

Figure 2.17: Freeloader architecture

Manager or Benefactor depending on the operation. Before a Freeloader Client is able to read/write data, it needs to contact the Freeloader Manager for details on the nodes which it is able to read/write data to/from. The Freeloader Client receives pairs of values containing chunk id and the Benefactor id. The Freeloader Client is then able to route its storage request to the correct Benefactor. When retrieving data-sets, the Freeloader Client will issue requests for chunks in parallel, aggregating network transfer from Benefactors. Whilst retrieving chunks, the Freeloader Client assembles them and presents a stream of data to the application.

Benefactor hosts run a daemon which is responsible for advertising its presence to the Freeloader Manager whilst servicing requests from Freeloader Clients and the Freeloader Manager. The Benefactor utilises local storage to store chunks; chunks relating to the same data-set are stored in the same file. The Benefactor services operations to create and delete data-sets from the Freeloader Manager and put and get operations from the Freeloader Client. The Benefactor monitors the local host's performance allowing it to throttle its service so as not to impede the host's operation.

The Freeloader Manager component is responsible for storing and maintaining the system's meta-data. The meta-data includes chunk ids and their Benefactor

locations, replication, checksums for each of the chunks and the necessary data to support client side encryption. The Freeloader Manager is responsible for chunk allocation utilising two algorithms: round robin and asymmetric striping. The round robin approach consists of striping data evenly across Benefactors, but as resource availability will vary from Benefactor to Benefactor, the algorithm has been altered to bias Benefactors with more available storage. The asymmetric approach involves striping data across Benefactors and the Freeloader Client itself, storing part of the data set locally. A local/remote ratio determines the proportion of chunks which are to be stored locally and on remote Benefactors. The ratio which yields optimal performance, *roughly corresponds to the local I/O rate and aggregate network transfer rate from the remote Benefactors.* Although this ratio may result in optimal operation, constraints imposed by limited local storage may not permit this ratio.

Implementation

The TCP Protocol is used to transfer chunks between the Freeloader Client and Benefactor, due to its reliability and its congestion/flow control mechanisms it was deemed suitable for larger transfers. The rest of the communication between the components is performed in UDP, as the messages are short and bursty in nature. An application utilising storage services will need to call the Freeloader library which implements some of the standard UNIX file I/O functions.

Benchmarks show the capability of asymmetric striping to aggregate disk I/O performance up to network capacity. A machine with a local disk speed throughput of 30MB/Sec was able to attain approx 95MB/Sec whilst striping data across remote nodes. At the moment, Freeloader has not been released, although it is documented that the Freeloader Client library has been written in C and implements the standard I/O function calls. Otherwise, it is unclear what languages were used to develop the Benefactor and Freeloader Manager components.

Summary

Freeloader's target audience includes scientists engaged in high performance computing that seek an inexpensive alternative to storing data whilst providing performance

associated with a parallel filesystem. Freeloader is designed to accommodate a transient flow of scientific data which exhibits a write-once/read-many workload. In doing so, it utilises existing infrastructure to aggregate storage and network bandwidth to achieve a fast, inexpensive storage solution providing scientists with an alternative to more expensive storage solutions like SANs.

2.2.10 PVFS

PVFS [16] is a parallel filesystem designed to operate on Linux clusters. The authors identify an absence of production quality, high-performance parallel filesystem for Linux clusters. Without a high-performance storage solution, Linux clusters cannot be used for large I/O intensive parallel applications. PVFS was designed to address this limitation and provide a platform for which further research into parallel filesystems. PVFS is designed to operate within the bounds of an institution in a trusted environment.

Architecture

PVFS was designed with three operational goals in mind, (i) provide high-performance access and support concurrent read/write operations from multiple processes to a common file, (ii) provide multiple interfaces/API's, (iii) allow existing applications which utilise POSIX file I/O to utilise PVFS without the need to modified or recompiled. The PVFS architecture is designed to operate as a client-server system (Figure 2.18). There are three main components which make up the PVFS system: PVFS Manager, PVFS Client and PVFS I/O daemon. A typical cluster environment has multiple nodes dedicated to storage and computation. Nodes responsible for storage run the PVFS I/O daemon and nodes responsible for computation will have the PVFS Client installed. An extra node is dedicated to running the PVFS Manager.

The PVFS Manager is responsible for storing meta-data and answering location requests from PVFS Clients. Meta-data stored by the PVFS Manager include filenames and attributes such as file size, permissions and striping attributes

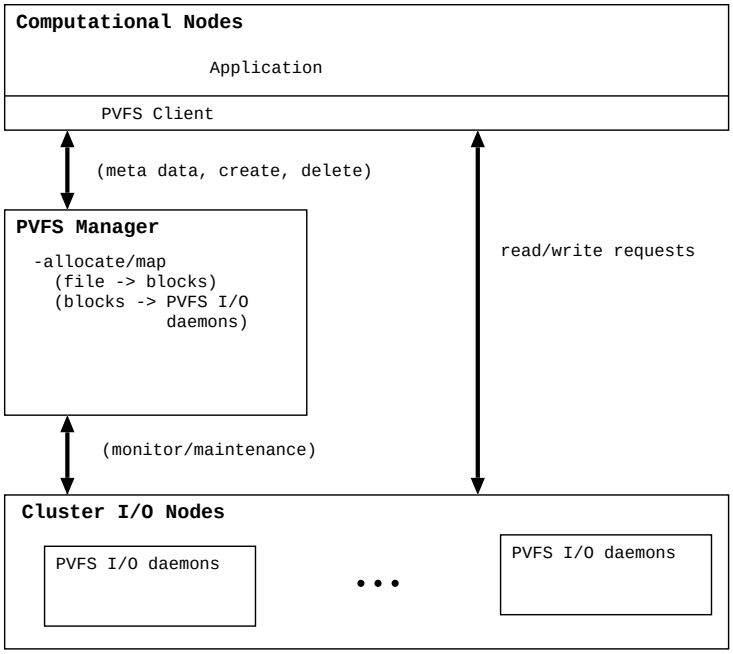

Figure 2.18: PVFS architecture

(segment size, segment count, segment location). The PVFS Manager does not service read/write requests, instead, this is the responsibility of the I/O daemon. Striping chunks of data across multiple I/O nodes allows parallel access. The PVFS Manager is responsible for enforcing a cluster wide consistent namespace. To avoid overheads associated with distributed locking and the possibilities of lock contention, PVFS employs a minimalistic approach to consistency with meta-data operations being atomic. Beyond enforcing atomic meta-data operations, PVFS does not implement any other consistency mechanisms. APIs provided by PVFS include a custom PVFS API, a UNIX POSIX I/O API and MPI-IO.

The PVFS Client is responsible for servicing storage requests from the application. Upon receiving a storage request, it will contact the PVFS Manager to determine which PVFS I/O daemons to contact. The PVFS Client than contacts the PVFS I/O daemons and issues read/write request. The PVFS Client library implements the standard suite of UNIX POSIX I/O API and when in place, traps any system I/O calls. The PVFS Client library than determines if the call should be handled by itself, or passed onto the kernel. This ensures that existing applications need not be modified or recompiled. The PVFS I/O daemon is responsible for

servicing storage requests from PVFS Clients whilst utilising local disk to store PVFS files.

Implementation

PVFS is distributed as open source under the GPL agreement and is available for download[7]. All components have been developed using C. PVFS uses TCP for all its communication so as to avoid any dependencies on custom communication protocols. Benchmarks conducted with 32 I/O daemons and 64MB files have shown to achieve 700MB/Sec using Myrinet and 225MB/Sec using 100Mbits/Sec Ethernet. PVFS is in use by the NASA Goddard Space Flight Centre, Oak Ridge National Laboratory and Argonne National Laboratory.

Summary

PVFS is a high-performance parallel filesystem designed to operate on a Linux clusters. It provides an inexpensive alternative utilising Commodity Off The Shelf (COTS) products allowing large I/O intensive applications to be run on Linux clusters. Benchmarks provided indeed show that PVFS provides a high-performance storage service. Some future work identified include a migration away from TCP, as it is deemed to be a performance bottleneck. Other areas of future research include: scheduling algorithms for I/O daemons, benchmarks show a performance flat spot, potential for further tuning and replication.

2.3 Survey of markets in Distributed Storage Systems

In this section we survey distributed storage systems which apply a market model to manage various aspects of their operation. During the survey we shall observe how: MojoNation [151] applies a market based on pseudo currency to instill good behaviour, SAV [25] employs a barter model to preserve archives, Mungi [70] applies

[7]PVFS Homepage: http://www.pvfs.org/

a commodity market model to manage storage quota and how OceanStore [82] presents a case for a storage utility.

2.3.1 Mungi

Mungi[70] employs marketing principles to ensure that storage is fairly allocated amongst local users. One of the primary design goals behind the way that they share storage was to ensure that any user is unable to starve other users of storage. Each user is issued with a bank account and the system provides income using a *pay master* and collects rent for storage used via a *rent collector*. To ensure users cannot amass a large number of credits over time and allocate vast amounts of storage potentially starving other users of storage, Mungi employs the following tax equation.

$$\tau(\beta) = b\left[1 - exp\left(-\frac{\beta}{b}\right)\right] \tag{2.1}$$

where:

- b: balance maximum, an account may never exceed this value. Equivalent to a storage quota.
- β: account balance.

The tax equation ensures users' balance tapers off and has virtually no effect on balances which are much lower than income. The cost function used by the *rent collector* to collect a fee from each of the user accounts is:

$$\rho(\xi) = 1 + 4\rho\xi^2 exp\left(\frac{\xi}{1-\xi} - 1\right) \tag{2.2}$$

where:

- ξ: $\xi(0 \leq \xi \leq 1)$ storage utilisation $(\xi = 0 : empty; \xi = 1 : full)$
- ρ: a parameter used to determine how quickly storage costs increases in relation to system wide storage utilisation.

The cost of storage grows exponentially as storage is being used up preventing the system from ever running out of storage. As users' credit runs out they are forced

to free up used storage. By employing a commodity market model, Mungi is able to share storage services more efficiently than otherwise possible with a fixed quota system. A fixed quota system is rather inflexible, users are forced to free storage when they reach their allocated quota even though there is plenty of free storage available system wide. A quota system limits administrators to allocate quotas such that the total sum of all quotas does not breach system storage capacity. Whilst this approach guarantees that users will always be granted storage, if within their quota, it results in much storage being wasted as most users will not reach their quota. Alternatively, administrators may choose to over commit the quotas and whilst this has the potential for better utilisation it runs the risk of exhausting system wide storage resulting in users being denied storage even if they are within their quota.

There are two main advantages associated with the approach employed by Mungi, (i) users have the potential to use a large quantity of storage temporarily and (ii) the crediting and debiting is run periodically rather than for every operation as with a quota system. An interesting observation made by the authors of Mungi [70] was that employing a bidding process, carries a significant overhead as users are forced to *play the market*; a limiting factor.

2.3.2 Stanford Archival Repository Project

The Stanford Archival Repository Project [25, 26, 27] aims to provide a way for institutions such as libraries to remotely archive their data repositories to improve reliability. As these institutions all possess some level of local storage and require remote storage there is a *double coincidence of wants* [110], ideal for bartering. With this in mind, the Stanford Archival Repository Project applies a bartering market model, allowing institutions to barter amongst each other effectively, remotely archiving each other's repositories.

The process of trading involves institutions calling auctions when they require remote storage. When an institution (A) calls an auction it advertises the size of storage (R) it requires. Institutions are invited to submit bids, their bids consist of the amount of storage they require in return from institution (A). Therefore, the lower the bid, the less storage institution (A) is required to provide in return for the

barter to happen, making lower bids more attractive.

Cooper et. al. [26] proposes the following two algorithms under which this auction may take place: Collection Trading and Deed Trading. Collection Trading involves bartering storage based on collection size, hence for a trade to be successful, both sites must be able to store each other's selected collection. On the other hand, Deed Trading is based on available blocks of data, this allows for extra flexibility, but as a consequence is more complicated, requiring institutions to keep records of deeds. Deed Trading allows a collection to be stored across many deeds, effectively splitting the collection across many distributed sites, otherwise not possible in Collection Trading. Deed Trading recognises that some deeds may have available space, allowing institutions to sub divide and create another deed from the available space. Results from simulations comparing Collection and Deed Trading algorithms, show that Deed Trading is more efficient, achieving higher global reliability whilst using less space.

In a more recent study [27] the following four biding algorithms were investigated:

1. **FreeSpace:** A site bids more, the more free space it has, therefore as its local storage becomes scarce it bids less and tends to wins more auctions. The aim of this algorithm is to encourage trading as storage becomes scarce.

2. **UsedSpace:** A site bids more, the less space it has available. Under this bidding policy a site begins to bid low when local space is abundant, and begins by winning most of the auctions. The idea behind this algorithm is to hold on to local storage as it becomes scarce.

3. **AbundantCollection:** A site bids more as its collections become more abundant. The effect of this bidding policy allows the institution to win auctions when replicas are low and therefore help to replicate its rare collections. As these rare collections get replicated (and become *not as* rare) the site starts to bid higher as the requirement to win auctions and replicate collections is not as urgent.

4. **RareCollection:** A site bids more when its collections are rare. Event though

with this strategy a site will win fewer auctions, when it does so it will have access to a large amount of storage (due to the high bid) therefore being able to archive many collections.

Simulations conducted varied a local storage factor (F) whilst paying particular attention to the resulting Mean Time To Failure (MTTF). The local storage factor (F) determines how much available storage a local site has compared with the size of its collection. A high value of F indicates an abundance of storage. Results from the simulation comparing the above four algorithms show that not any one of the bidding policies was outright best for all values of F. For low values of F ($2 <= F <= 3.5$) FreeSpace achieves excellent results, when F is between ($3.5 < F <= 4.5$) all the algorithms perform fairly similarly and for high values of F ($4.5 < F <= 6$) UsedSpace becomes the dominant strategy.

The Stanford Archival Repository Project possesses unique requirements and qualities, users with common objectives, barter amongst each other for remote store, in the process preserving their collections. Each user has local store, which if made available to remote users is deemed valuable, its a situation that is ideal for bartering. Cooper et. al. investigate a framework for data trading, employing an economic mechanism structured around reliability, rather than access performance. The bartering model employed is symmetric and suits a Peer to Peer architecture where every site is autonomous, capable of providing archival storage as well as requiring it. The result of this investigation is a framework which allows institutions to replicate collections amongst each other effectively creating a global platform for data archiving.

2.3.3 MojoNation

MojoNation [151] is a peer-to-peer file sharing network functionally much like Gnutella and Napster. The major difference over earlier systems and its relevance to our investigation is its use of market principles to balance load on the network and instill good behaviour amongst users in an otherwise untrusted environment. Storage space, bandwidth and processing cycles are all services recognised by MojoNation

and translate to *Mojo*, a pseudo currency. When users contribute these services, they are issued with *Mojo* and when services are used they are charged.

Every service provided or consumed requires a transaction to be performed. To ensure honest transfers amongst peers, a trusted third party entity (Broker) is employed. The Broker oversees user transactions and behaviour. Any interaction amongst peers, be-it a hello message or request for a service will result in an offer of *Mojo* being made. The initiating peer offers *Mojo* and the receiving agent (A) extends the initiating peer credit in order to complete the transaction. When the credit limit is reached, receiving agent (A) contacts the Broker to complete the transaction. Interactions with the Broker are limited to either when a credit limit is reached or when a transaction sums up to a *coin*. There are two benefits for limiting interactions with the Broker: (i) peers are able to continue to function even if the Broker is temporarily unavailable and (ii) reduce the load on the Broker, allowing it to service more peers. The Broker is responsible for maintaining accounts and balances and establishes trust by overseeing transactions whilst keeping account of reputation.

The market model employed by MojoNation is based on a brokering model where pseudo currency is used as a temporary medium to exchange services amongst peers. Whilst communication and capabilities of peers corresponds to what is typical of a peer-to-peer system, the Broker introduces centralisation, which has been recognised as a source of performance and scalability issues.

2.3.4 OceanStore

OceanStore [82] proposed an architecture for a globally scalable storage utility, whereby consumers would pay a fee in exchange for access to persistent storage. Providing a storage utility inherently means that data must be highly available, secure, easy to access and provide guarantees on Quality of Service (QoS). A user must be able to access their data easily from any geographic location. Hence data must be cached in geographically distant locations, in effect travelling with the user, and giving rise to the term *nomadic* data. OceanStore provides a mount point providing users with a transparent, easy to use interface, hiding all the underlying

complexities, enabling existing applications to access storage.

OceanStore aims to provide high performance and availability guarantees whilst operating across a global untrusted infrastructure. Providing a storage utility implies the need for accountability mechanisms to be employed, a pricing scheme and payment system. Keeping track of transactions and managing accountability within a distributed untrusted environment is a challenging task and requires a trusted platform to function. The OceanStore architecture has the potential to implement a market model within the Byzantine set of nodes, unfortunately no details are supplied.

2.4 Discussion and Summary

We have presented a taxonomy of distributed storage systems including a survey of systems which apply market models to manage various aspects of their operation. In our study we have found that distributed storage systems are evolving, providing richer functionality (Section 2.1.1), operating across untrusted environments with tougher constraints (Section 2.1.3), adopting more scalable ad-hoc architectures (Section 2.1.2), employing dynamic routing (Section 2.1.9), optimistic consistency (Section 2.1.5) and cryptographic algorithms to provide security (Section 2.1.6). With the emergence of many different distributed storage systems, federating and managing globally distributed data is becoming an increasingly challenging task, sparking research into Data Grids (Section 2.1.8). All these factors are making DSSs increasingly complex and consequently harder to maintain and administer.

This complexity dilemma has been identified as one of the toughest hurdles facing computer systems [80] and to address this we have seen the emergence of Autonomic Computing (Section 2.1.7). Autonomic computing has inspired much innovative research proposing many unique ways to address issues relating to complexity; from structured methods like introspection [82] and the four axioms of self configuration, optimisation, healing and protection [72] to more ad-hoc approaches inspired by biological [134] and economic [52, 152, 15] systems. The focus in our thesis is on distributed storage systems which apply economic principles to manage various

aspects of their operation. A survey (Section 2.3) of distributed storage systems systems that apply market models to manage various aspects of operation was presented, in summary we discussed: quota management (Mungi [70]), encouraging the sharing of storage for data preservation (SAV [25]), instilling cooperative behaviour (MojoNation [151]), and provision of a global storage utility (OceanStore [82]). It is in this context that we propose the Storage Exchange platform.

Essentially, the Storage Exchange [104] platform allows storage to be treated as a tradeable resource. Consumers and providers are able to submit their storage requirements and services along with budgetary constraints to the Storage Exchange, which in turn employs a market model to determine successful trades. The Storage Exchange encourages the sharing of unused storage services much like SAV but is more dynamic allowing storage services to not only be bartered but traded as storage utility. As a utility, the Storage Exchange can be compared to OceanStore. Whilst OceanStore provides many invaluable insights into providing a reliable global storage service, it shies away from applying a market model enabling storage services to be automatically traded. Whilst the Storage Exchange can be likened to OceanStore or SAV, it has been designed to be a platform for future research into autonomic management of storage services. We envisage consumers and providers will employ brokers to automatically trade storage based upon organisational requirements. The Storage Exchange is discussed in detail in the next chapter.

Chapter 3

STORAGE EXCHANGE PLATFORM

In this chapter we discuss the Storage Exchange platform in detail. We begin with a system overview where we introduce the Storage Exchange and discuss various roles an institution may choose to adopt when using our platform. We then present the architecture, where we introduce the Storage Provider, Storage Client, Storage Broker and Storage Marketplace. Upon discussing each of the main components, we introduce Virtual Volumes and how they can be traded and utilised. This is followed by a series of sections dedicated to each of the main components covering architectural and design details. The chapter continues by providing an insight into implementation, before discussing our evaluation and subsequent performance results. We conclude by summarising the main aspects of the platform.

3.1 Introduction

The Storage Exchange [104] is a platform allowing storage to be treated as a tradeable resource. Organisations with available storage are able to use the Storage Exchange to lease it out to consumer organisations. The Storage Exchange platform has been designed to operate on a global network such as the Internet, allowing organisations across geographic and administrative boundaries to participate.

"Commerce in every era consists of sellers finding buyers at mutually beneficial prices" [17]. The Storage Exchange platform is no different, providing consumers and providers with a place to advertise their requirements whilst employing a market model to efficiently allocate trades which are mutually beneficial. There are three important functions which make the Storage Exchange platform a reality; (i) the ability to harness available storage (Storage Provider), (ii) provide an interface to the storage (Storage Client), (iii) manage and trade the storage (Storage Broker and

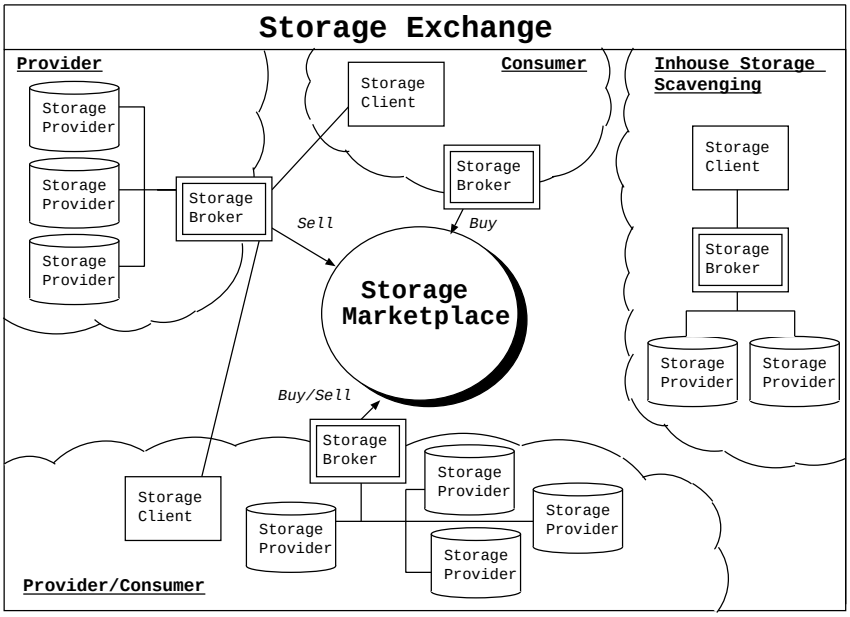

Figure 3.1: Storage Exchange: platform overview

Storage Marketplace). The Storage Exchange has many applications, to illustrate we discuss the possible roles an organisation may choose to adopt when using our platform (Figure 3.1):

1. **Provider:** Organisations with an abundance of storage may participate as providers selling their available storage, consequently better utilising their existing infrastructure.

2. **Consumer:** Organisations which require storage beyond the capacity of internal storage services may participate as consumers and purchase the storage services. Purchased storage may be used for remote archival, day to day file store or just temporary storage.

3. **Provider and Consumer:** Organisations adopting this role actively buy and sell storage services. Some possible reasons for adopting this role include:

   (a) *Barter:* Organisations choosing to barter aim to sell storage to cover expenses accrued from purchasing storage. Participants adopting this role need not pay to participate. If all participants were configured to barter, the system would operate under similar principles to the Standford

Archival Repository Project [25], where institutions barter with each other to archive each other's repositories.

(b) *Temporary access to storage:* Some organisations may require access to storage services temporarily, an example scenario would be to briefly store results generated from scientific simulations [143]. To cover these spikes in storage requirements, organisations mostly behave as providers, providing storage services and acquiring credits, then when they require storage they have the ability to purchase storage with the acquired credits. This behaviour allows an organisation to temporarily access storage services beyond their own without the need to pay.

4. **In-house Storage Scavenging:** In this configuration, an organisation chooses not to communicate with the Storage Marketplace and instead meets their storage requirements by utilising available storage within the organisation. In this configuration the Storage Exchange platform would be functionally similar to that of the Farsite [2] system.

All roles except *In-house Storage Scavenging* need to interact with the Storage Marketplace. From a computer systems view point, the Storage Marketplace is an entity responsible for resource discovery, allowing consumers to find resources. From an economic angle, it allows providers to find consumers. In our discussion thus far, we have seen the possible uses of the Storage Exchange. The following sections will focus on its architecture and individual components.

3.2 System Architecture

In this section we shall introduce the basic ideas which allow the Storage Exchange platform to function. We begin by briefly discussing each of the four main components (Storage Provider, Storage Client, Storage Broker and Storage Marketplace), followed by a discussion of Virtual Volumes and the process of utilising and trading them.

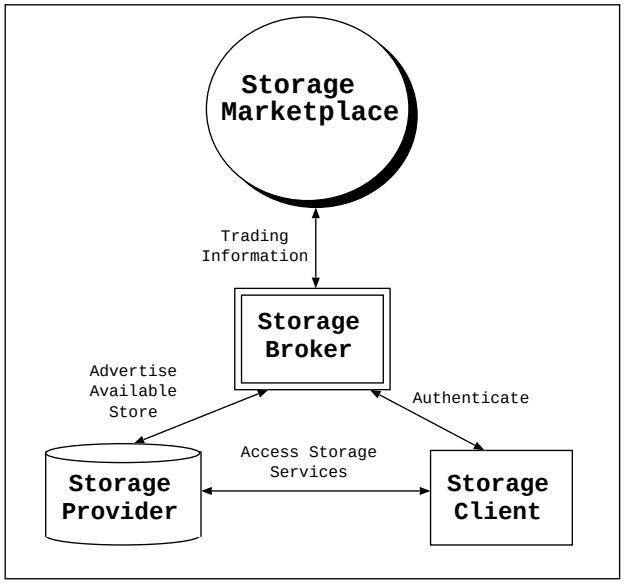

Figure 3.2: Storage Exchange: architecture

There are four main components which make up the Storage Exchange platform (Figure 3.2): (i) Storage Provider: harnesses available storage on installed host whilst servicing requests from Storage Client, (ii) Storage Client: provides an interface for the user to access storage services, (iii) Storage Broker: manages in-house storage capacity and trades storage based upon storage service requirements of institution, (iv) Storage Marketplace: a trading platform used by Storage Brokers to trade storage. The Storage Marketplace and Storage Broker are mainly responsible for exchanging trading information, the Storage Client and Storage Provider communicate with each other when storage services are being accessed. Both the Storage Provider and Storage Client communicate with the Storage Broker, the Storage Provider does so to inform of storage usage and the Storage Client is required to authenticate itself with the Storage Broker before being able to access storage services.

3.2.1 Storage Provider

The Storage Provider is deployed on hosts within an organisation chosen to contribute their available storage. Whilst we envision the Storage Provider to be used to scavenge available storage from workstations, there is no reason why it

cannot be installed on servers or dedicated hosts. The Storage Provider is responsible for keeping the organisation's broker up to date with various usage statistics and servicing incoming storage requests initiated by Storage Clients.

3.2.2 Storage Client

A Storage Client enables an organisation to utilise storage services, be it internally or from an external organisation. The user needs to configure the Storage Client with their user credentials, storage contract details and the Storage Broker it should contact. The Storage Client then transmits the user and contract details to the specified Storage Broker. Upon successful authentication, the Storage Broker looks up the Storage Providers responsible for servicing the storage contract and instructs them to connect to the Storage Client. Once the Providers establish a connection with the Client, the Client then provides a filesystem like interface, much like an NFS [121] mount point. The filesystem interface provided by the Storage Client allows applications to access the storage service like any other file system and therefore applications need not be modified or linked with special libraries.

3.2.3 Storage Broker

For an organisation to be able to participate in the Storage Exchange platform they will need to use a Storage Broker. The Broker enables the organisation to manage their available storage, buy and sell storage services and allow Storage Clients to utilise storage services. The Broker needs to be configured to reflect how it should best serve the organisation's interests. From a consumer's perspective, the Broker will need to know the organisation's storage requirements and the budget it is allowed to spend in the process of acquiring them. From the Provider's perspective the Storage Broker needs to be aware of the available storage and the financial goals it is required to reach. Upon configuration, a Storage Broker contacts the Storage Marketplace with its requirements.

The Storage Broker is the largest component and is responsible for:

1. Keeping track of bought and sold storage contracts.

2. Monitoring system activity, covering status of Storage Providers and storage contract usage.

3. Authenticating both internal and external Storage Clients.

4. Maintaining routing information; mapping storage services to Storage Providers.

The Storage Broker uses statistics received from Storage Providers to make decisions on how best to manage the infrastructure (e.g. allocate storage amongst Storage Providers) to ensure continuous operation. These statistics can be used to determine which providers are deemed unavailable, and can be used to maintain a level of redundancy, increasing availability whilst reducing the risk of losing data.

3.2.4 Storage Marketplace

The Storage Marketplace provides a platform for Storage Brokers to advertise their storage services and requirements. The Storage Marketplace is a trusted entity responsible for executing a market model and determining how storage services are traded. When requests for storage are assigned to available storage, the Storage Marketplace generates a storage contract. The storage contract will contain a configuration of the storage policy and form a contract binding the provider to fulfill the service for the determined price. In a situation where either the provider or consumer breaches a storage contract, the Storage Marketplace has the potential to keep a record of reputation for each organisation which can be used to influence future trade allocations.

3.2.5 Virtual Volume

A Virtual Volume refers to a storage device a Storage Client accesses when communicating with a Storage Provider. A Storage Client only communicates with a single Storage Provider (Primary) who is responsible for providing a single homogeneous interface to the Virtual Volume. A Virtual Volume has the potential to be distributed across multiple Storage Providers and it is the responsibility of the

Primary Storage Provider to service Storage Client requests by routing them to the relevant Secondary Storage Providers (Figure 3.3). Our implementation of Virtual Volumes supports replication, that is a Virtual Volume can be replicated across multiple Storage Providers, ensuring better reliability and availability in the face of outages. Other future possibilities would be to support various modes of striping to improve performance, erasure codes for more efficient replication or Distributed Hash Tables (DHT) for their excellent distribution properties.

To support a replicated Virtual Volume, the Primary Storage Provider is required to establish connections to all the Secondary Storage Providers containing replicas. When the Primary Storage Provider receives a write operation it needs to route it to all the Secondary Storage Providers ensuring all replicas remain consistent. In Figure 3.3 we can see Virtual Volume A and B, both have been configured with three way replication. Storage Provider(1) is currently behaving as the Primary for Virtual Volume A and a Secondary for Virtual Volume B. If a Primary Provider is to fail, a Secondary Storage Provider can take its place as the Primary and service for that Virtual Volume may resume. Storage Providers have the ability to service multiple Volumes and in doing so are able to adopt different roles for each. As any Storage Provider with a replica can be chosen to be the Primary Storage Provider, improved load balancing is achieved. A possible future extension would be to allow Primary Storage Providers to store replicas on Secondary Storage Providers outside the organisation. Replicating across organisations would provide offsite redundancy, allowing data to be accessed even in the event an organisation's network were to be made unavailable.

Our implementation of the Storage Provider has been designed to adopt a strong consistency methodology, therefore the Primary Provider only notifies the Storage Client of a successful write operation if the write operation was successfully executed on all replicas. Whilst such a pessimistic approach to consistency decreases performance and availability, it reduces the risk of inconsistencies arising amongst replicas. A future development would be to support a more optimistic approach to consistency and provide automated recovery mechanisms to detect inconsistencies and resolve them.

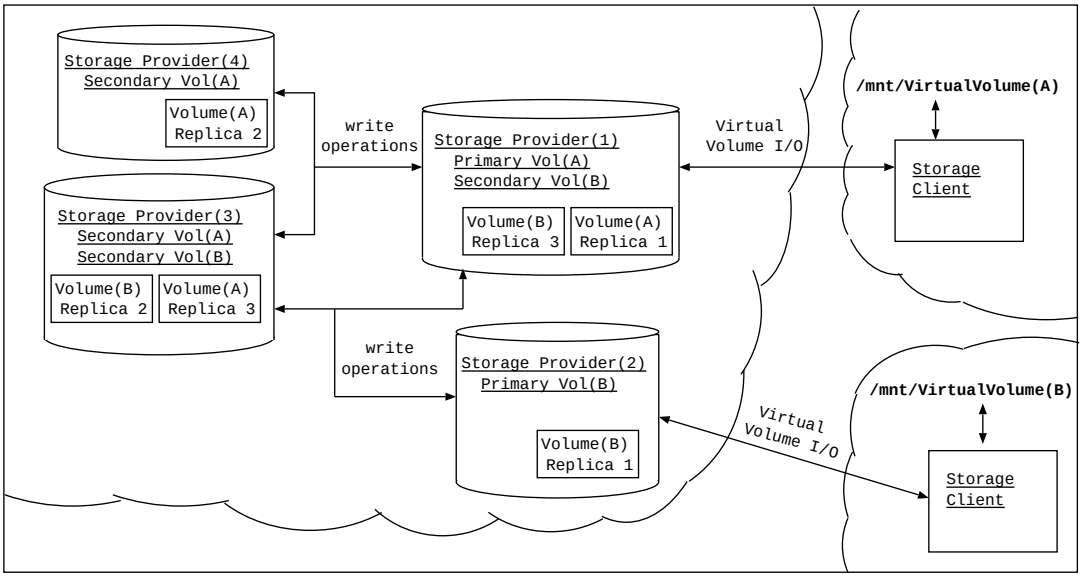

Figure 3.3: System architecture: virtual volume

3.2.6 Mounting a Virtual Volume

Before a user is able to utilise a Virtual Volume, they first need to mount it. In this section we shall discuss this process. It is assumed the user requiring access to the Virtual Volume has either purchased a storage contract from the hosting institution or the Virtual Volume is hosted within the institution by that user. The user needs to configure the Storage Client with authentication details and the Virtual Volume they wish to mount. The process of mounting a volume can be broken down into the following four steps (Figure 3.4):

1. *Request to Mount:* Storage Client sends a mount request to the institution's Storage Broker which is responsible for hosting the Virtual Volume.

2. *Service Volume:* Storage Broker than queries its database, ensuring the Storage Client has the correct credentials. Upon successful authentication, it queries the database for the IP addresses of the Storage Providers allocated to service the Virtual Volume. The Storage Broker than selects a Storage Provider to be the Primary and relays the Storage Client request along with the IP address of the Storage Client and participating Secondary Storage Providers.

3. *Build Volume:* Upon receiving the request to service the volume from the

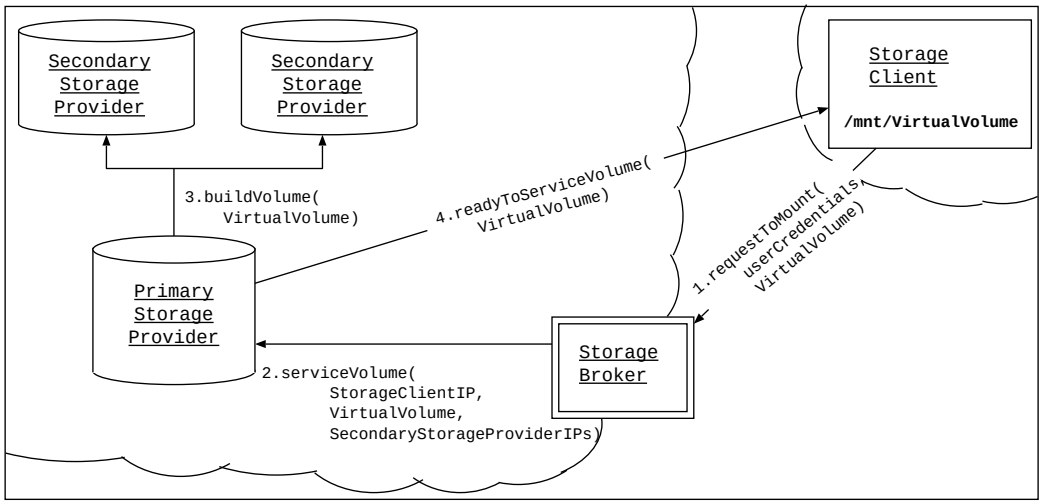

Figure 3.4: System architecture: mounting a virtual volume

Storage Broker, the Primary Storage Provider establishes connections to the specified Secondary Storage Providers.

4. *Ready to service volume:* Once the Primary Provider receives connections from all Secondary Providers and it is satisfied it may service the Virtual Volume, it initiates a connection to the Storage Client which originally requested the mount of the Virtual Volume. Once the Primary Storage Provider establishes contact with the Storage Client, the user is then able to utilise the Virtual Volume.

The process of mounting a Virtual Volume has been designed to limit incoming connections to the Storage Broker. This allows Storage Providers to operate behind a firewall configured to block incoming connections. Hence an institution need only configure their firewall to only allow incoming connections to their Storage Broker IP address and port number. The Storage Broker is the institution's gateway, a single point of access to the outside world allowing intuitions to participate in the Storage Exchange platform. Although having the Storage Broker as the gateway has its disadvantages, particularly being a Single Point of Failure (SPF), it allows the institution to participate securely without undermining established security mechanisms.

3.2.7 Trading Virtual Volumes

In this section we discuss the process behind trading Virtual Volumes. We begin by outlining Storage Policies; which provide a way to quantify storage being traded. Our discussion then continues with a look at how the Storage Broker and the Storage Marketplace communicate to permit the trading of storage.

A Storage Broker uses Storage Policies to quantify the service which they wish to lease or acquire. When a trade is determined, the storage policy will form the basis for a storage contract containing details of the SLA (Service Level Agreement). The Storage Policy (SP) used by the Storage Marketplace is defined as $SP = (C, U, D, T)$ where:

C : Storage Capacity (GB) of volume.
U : Upload Rate (KB/sec).
D : Download Rate (KB/sec).
T : Duration.

Storage Brokers submit bids and asks, each containing a SP, to the Storage Marketplace, which accepts these bids, and allocates trades by applying a market model discussed in chapter 4. The trading process can be broken down into the following events (Figure 3.5):

1. **SRB:** Storage Brokers wishing to purchase storage do so by submitting a Storage Request Bid (SRB). A SRB contains a SP detailing the service to be purchased along with a bid price $SRB = (SP_{SRB}, \$)$.

2. **SSA:** A Storage Broker wishing to sell storage may do so by submitting a Storage Service Ask (SSA). An SSA also contains a SP which details the storage service being sold, along with a cost function $SSA = (SP_{SSA}, CostFunction(SP_{SRB}))$. The *CostFunction* is used by the Storage Marketplace to determine the ask price for a service configuration specified by an SRB's SP (SP_{SRB}). This allows services provided by a SSA to be configured to the specific requirements of an SRB. It also allows a single SSA to potentially service multiple SRBs, assuming it has the capacity (large SP attributes).

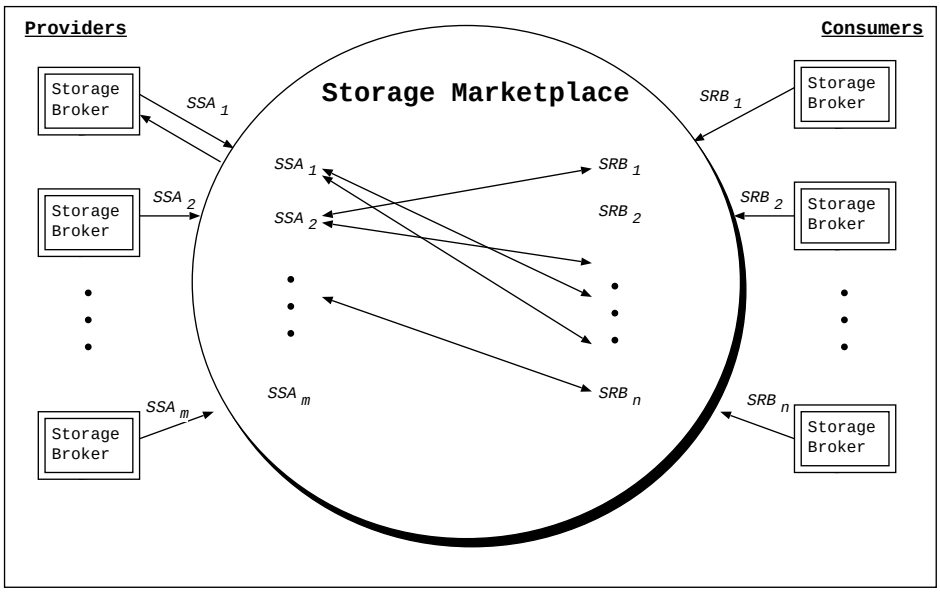

Figure 3.5: System architecture: trading virtual volumes

3. **Allocating Trades:** The Storage Marketplace applies a market model to allocate trades and notifies all the participating Storage Brokers of the outcome. Storage Brokers that submitted successful SRBs are supplied with the details of the Provider's Storage Broker along with a Virtual Volume identifier. Storage Brokers which submitted successful SSAs are notified of all the storage services they will need to host.

The rest of this chapter discusses each component within the Storage Exchange platform in greater detail, covering architecture and design. A section discussing implementation details, is followed by performance evaluation and concludes with a summary leading to our next chapter on market models.

3.3 Storage Provider

The Storage Provider is responsible for servicing storage requests from Storage Clients and does so by utilising locally available storage. Whilst servicing Storage Clients, the Storage Provider is also responsible for reporting back storage usage statistics to the Storage Broker. Upon successful installation, the Storage Provider needs to be configured with the address of the institution's Storage Broker, along

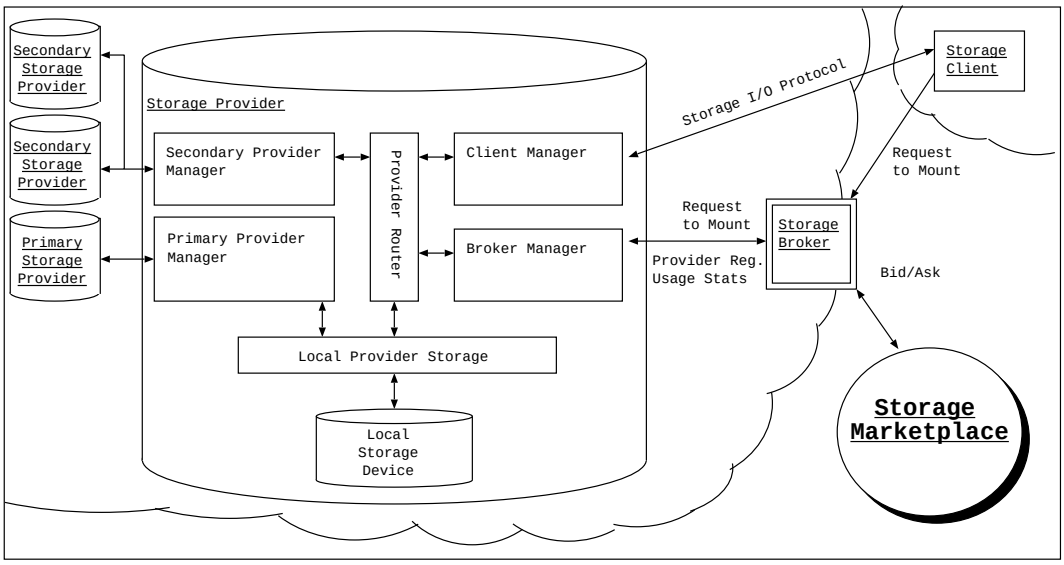

Figure 3.6: Storage Provider: component diagram

with a registered user account.

3.3.1 Architecture

The Storage Provider architecture is based on six main components (Figure 3.6); Client Manager, Broker Manager, Provider Router, Primary Provider Manager, Local Provider Storage, Secondary Provider Manager. We discuss each in detail.

1. **Client Manager:** The Client Manager is responsible for initiating an outbound connection to the Storage Client as part of the process of mounting a Virtual Volume (Section 3.2.6: Step 4). Once a connection is established, the Storage Provider is able to service storage requests from the Storage Client. The storage protocol used between the Storage Provider and Storage Client is described in Appendix B.2.3.

2. **Broker Manager:** When the Storage Provider is first executed, the Broker Manager contacts the specified Storage Broker presenting the user account. Upon successful authentication, the Storage Broker will issue the Broker Manager with a unique identifier (Storage Entity ID). The Broker Manager will then use that Storage Entity ID for every subsequent sign-on to the Storage Broker. During a sign-on, the Broker Manager is also responsible for

reporting current storage capacity and usage statistics to the Storage Broker. The handshake used by the Broker Manager to sign-on with the Storage Broker is described in Appendix B.2.1.

The Storage Broker uses these sign-on messages to gauge availability and storage capacity of all Storage Providers. This information allows the Storage Broker to determine which Storage Provider is available to be the Primary Provider and service a Storage Client's requests. The reported storage capacity is also used by the Storage Broker when allocating Storage Providers to newly created Virtual Volumes. The Broker Manager component may also receive a request from the Storage Broker to service a volume (Section 3.2.6: Step 2). This request is simply passed onto the Provider Router component.

3. **Secondary Provider Manager:** Is used by the Storage Provider to initiate connections to Secondary Storage Providers. The Secondary Provider Manager receives storage requests from the Provider Router component and relays them to be serviced by Secondary Storage Providers. In response to these storage requests, the Secondary Storage Providers send replies to the Secondary Provider Manager which are forwarded to the Provider Router.

4. **Local Store Manager:** This component is responsible for using local file I/O system calls to service incoming storage requests from either the Primary Provider Manager or Provider Router. The Local Storage Manager is configured with a local directory to use for particular storage. Each virtual volume is assigned a directory and the Local Store Manager ensures that each volume is sand-boxed in their respective directory and cannot effect other volumes or be used to access other files on the host.

5. **Provider Router:** The Provider Router is the core component in the Storage Provider architecture, responsible for receiving storage events (Appendix B.1) and routing them to the corresponding components. The Provider Router may receive events from the following components:

   (a) *Broker Manager:* A request to service a volume may be received from

the Broker Manager. If it is the first time this volume is mounted, the Provider Router contacts the Local Store Manager to create a directory for the volume, effectively sand-boxing any storage requests for the volume to that directory. If the request to service details secondary providers, the Broker Manager sends events to the Secondary Provider Manager to establish connections to the necessary Secondary Providers. Once the Provider Router is satisfied that it may service the volume, it sends an event to the Client Manager to initiate a connection to the Storage Client that requested to mount that volume.

(b) *Client Manager:* Storage requests issued by the Storage Client are received by the Client Manager which relays them to the Provider Router. Depending on the type of storage request, the Provider Router needs to adopt a different approach. To illustrate (Figure 3.3), if the storage request is a read-only type operation, the Storage Provider need only route it to the Local Store Manager, otherwise if it is a write operation it will need be executed on the other replicas and thus is forwarded to the Secondary Provider Manager.

(c) *Secondary Provider Manager:* The Secondary Provider Manager relays events from Secondary Storage Providers to the Provider Router. The Provider Router uses these replies to ensure storage requests have been successfully executed and can relay, via the Client Manager, a corresponding reply back to the Storage Client.

(d) *Local Store Manager:* The Local Store Manager sends replies back to the Provider Router for storage requests it has serviced.

6. **Primary Provider Manager:** The Primary Provider Manager is responsible for accepting connections from Primary Providers. When a Primary Provider Manager accepts a connection it means that it will behave as a Secondary Storage Provider when servicing requests for that volume.

3.3.2 Design

The Storage Provider is a multi-threaded application that was developed using the C language. Every thread, including the types of messages relayed amongst the threads are detailed (Figure 3.7). The Storage Provider's design is consistent with our architecture, with each component in the architecture translating to a thread(s) in the design. To ensure data structures remain consistent whilst being passed between threads, we employ thread safe fifos. The Client Manager, Primary Provider Manager and Secondary Provider Manager have been designed to spawn threads for every connection, allowing storage events to be serviced from multiple Storage Clients, Primary Storage Providers and Secondary Storage Providers concurrently. The Storage Provider is able to service multiple Virtual Volumes concurrently.

The Provider Router sits at the core of the design and is responsible for handling all incoming storage events, with the exception of events emanating from Primary Storage Providers. Algorithm 1 shows the manner by which the Provider Router thread processes incoming storage events. The Provider Router ensures that if a storage event modifies the Virtual Volume, it is relayed to all Secondary Storage Providers storing replicas. Before a reply is sent back to the Storage Client the Provider Router must receive replies from each of the Secondary Storage Providers.

3.4 Storage Client

The Storage Client is responsible for providing a mount point interface to the underlying Virtual Volume. The Virtual Volume can be hosted by an external institution, requiring the Storage Client contact that institution's Storage Broker in the process of mounting it (Section 3.2.6).

3.4.1 Architecture

The Storage Client architecture is made up of four main components (Figure 3.8), the Storage Broker Manager, File Interface Manager, Provider Manager and the Client Router.

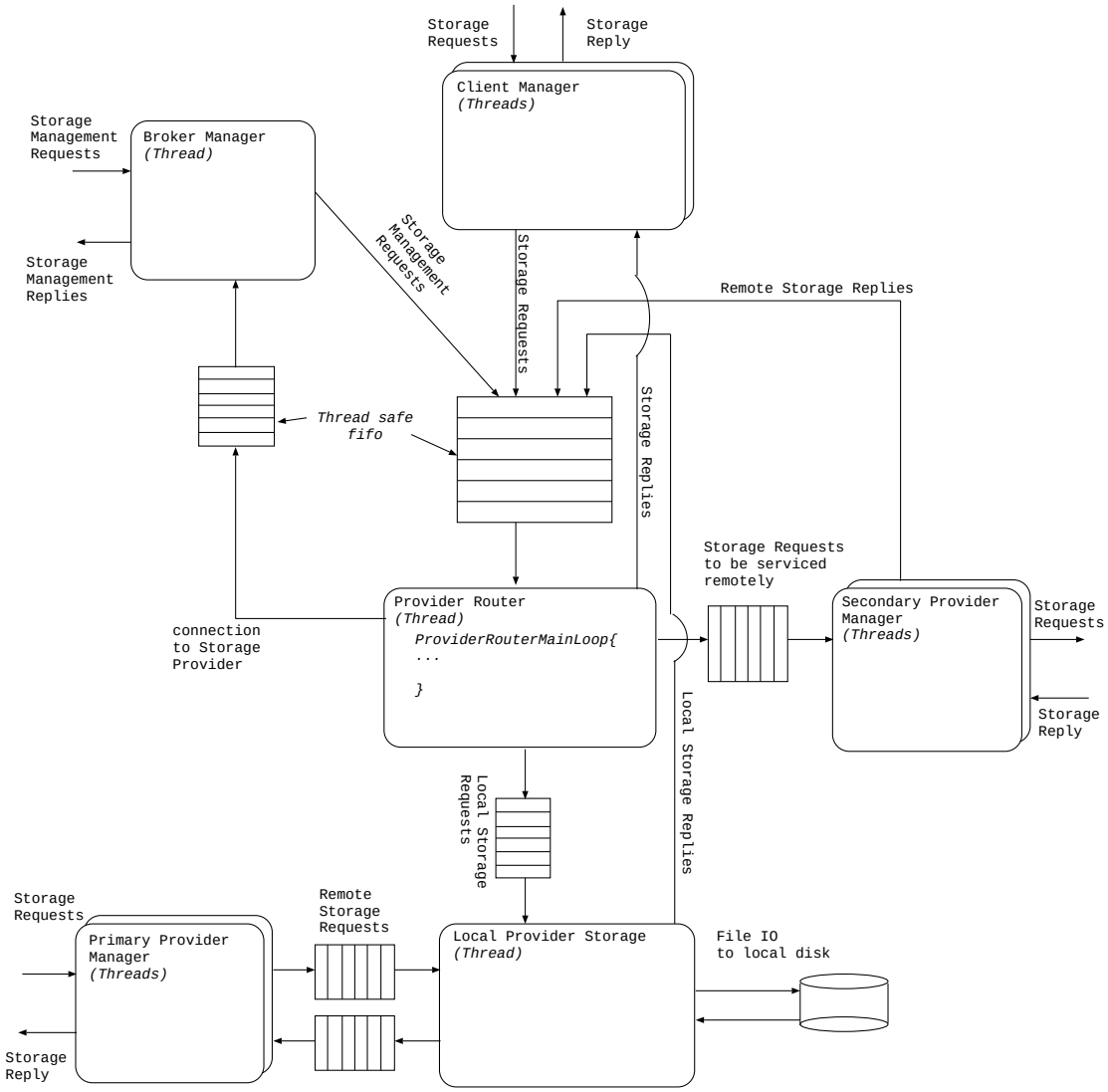

Figure 3.7: Storage Provider: threading and message passing

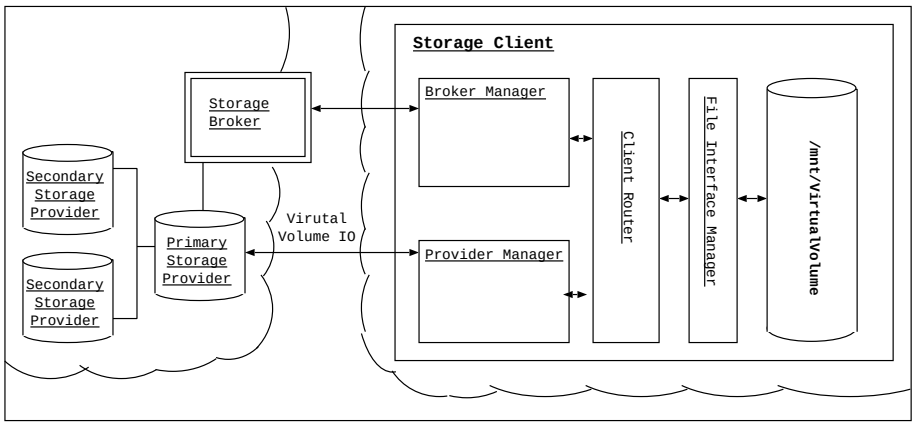

Figure 3.8: Storage Client: component diagram

Algorithm 1 Provider Router Main Loop

```
1: Input: Incoming storage events on thread safe fifo ISE
2: Output: Local Provider Storage thread safe fifo LPS,
   Remote Secondary Storage Provider thread safe fifo, SSP
   Client Manager thread safe fifo CM
3: PSR ← {∅} // a set used to keep track of pending storage requests
4: for all StorageEvent ∈ ISE do
5:   if StorageEvent is of type file IO then
6:     if StorageEvent is a request from a client then
7:       PSR ← R ∪ StorageEvent
8:       if StorageEvent is a read-only storage request then
9:         if StorageEvent can be serviced locally then
10:          LPS ← LPS ∪ StorageEvent
11:        else
12:          sendToOneSecondaryStorageProvider(SSP, StorageEvent)
13:        end if
14:      else if StorageEvent is a write request then
15:        multiCastToAllSecondaryStorageProvider(SSP, StorageEvent)
16:      end if
17:    else if StorageEvent is a reply then
18:      OSR ← {∅} // Original storage request
19:      OSR ←findTheOrigRequestThisReplyIsFor(PSR, StorageEvent)
20:      if StorageEvent is a reply to read-only request ∨
         replies received from all Providers then
21:        PSR ← PSR \ OSR // remove from pending list
22:        CM ← CM ∪ StorageEvent // put reply on queue to be sent to client
23:      end if
24:    end if
25:  else if StorageEvent is of type management request then
26:    if StorageEvent is a request for provider then
27:      SSPC ← {∅} // Secondary Storage Providers to Connect to
28:      SSPC ← getAllSecondaryProvidersConnDetails(StorageEvent)
29:      establishConnectionsToSecondaryProviders(SSPC)
30:    else if StorageEvent is notify of secondary provider connection then
31:      if All necessary Secondary Storage Providers connected for volume then
32:        connectToClientVolumeIsReady()
33:      end if
34:    end if
35:  end if
36: end for
```

1. **Broker Manager:** When the user starts the Storage Client, the Broker Manager is responsible for initiating a connection and sending a request to mount (Section 3.2.6: Step 1) to the Storage Broker managing the Virtual

Volume.

2. **File Interface Manager:** The File Interface Manager interfaces to the FUSE [56] kernel module. FUSE is an open source effort, allowing users to develop and mount file systems in user space. The File Interface Manager complies with the FUSE API, which closely resembles the file system I/O calls. When an application accesses the FUSE file system, the FUSE kernel module executes functions within the File Interface Manager which implement the API. The File Interface Manager then translates these calls to Virtual Volume storage events and relays them to the Provider Manager to send to the Storage Provider. The protocol that is subsequently used by the Provider Manager to communicate with the Storage Provider is based upon the FUSE API (Appendix B).

3. **Provider Manager:** The Provider Manager is responsible for sending storage requests to the Storage Provider, who services these requests and sends replies back to the Provider Manager. It is the responsibility of the Storage Provider to establish a connection with the Provider Manager, as part of the mounting process (Section 3.2.6: Step 4).

4. **Client Router:** The Client Router sits at the core of the Storage Client architecture. It is responsible for processing incoming storage events and routing them to the correct components.

3.4.2 Design

Like the Storage Provider the Storage Client is a multi-threaded application that was developed using the C language. Every thread, including the types of messages relayed amongst the threads are detailed (Figure 3.9). The Storage Client's design closely follows its architecture. There are a few details the architecture hides, these include the MonitorPendingStorageRequests thread and how the File Interface Manager module interfaces with FUSE:

1. **File Interface Manager:** Each time an application accesses the Virtual Volume mount point, a call is made to the Virtual File System (VFS) kernel

module, which in turn routes these file operations to the FUSE kernel module to process. The FUSE kernel module than starts a Light Weight Thread (LWT), executing a function in the File Interface Manager that is equivalent to the file operation. The File Interface Manager generates a request storage event which is registered with MonitorPendingStorageRequests and is placed on the Client Router thread safe fifo before blocking the Light Weight Thread and waiting for a reply. Upon receiving the reply storage event, the Client Router wakes the blocked Light Weight Thread passing it the reply storage event. Upon waking and receiving the reply storage event, the Light Weight Thread removes the corresponding requesting storage event from the pending list, retrieves data from the reply storage event and returns to the FUSE module. The FUSE module than relays this information to the VFS which presents it to the application that was accessing the Virtual Volume mount point.

2. **MonitorPendingStorageRequests:** This thread is responsible for monitoring a list of pending storage requests which are waiting to be serviced. Storage requests added to the list are assigned a configured retry time out. Every second the MonitorPendingStorageRequests traverses the list of pending storage requests decrementing the retry timers. Pending storage requests whose timer reaches 0 are reissued to the Client Router to be processed again and their retry timeout is reset. This ensures that if a storage request is lost (e.g. due to loss of connectivity with the Primary Storage Provider) the storage request is re-issued. As each requesting storage event is assigned a unique identifier, multiple requesting storage events can be executed in parallel, allowing multiple applications accessing the Virtual Volume to be serviced in parallel. The unique identifier also ensures that duplicate requests received by the Primary Storage Provider are ignored.

The Client Router is positioned at the core of the Storage Client design and receives all incoming storage events. Algorithm 2 details how the Client Router processes these incoming storage events and how it manages losing connectivity

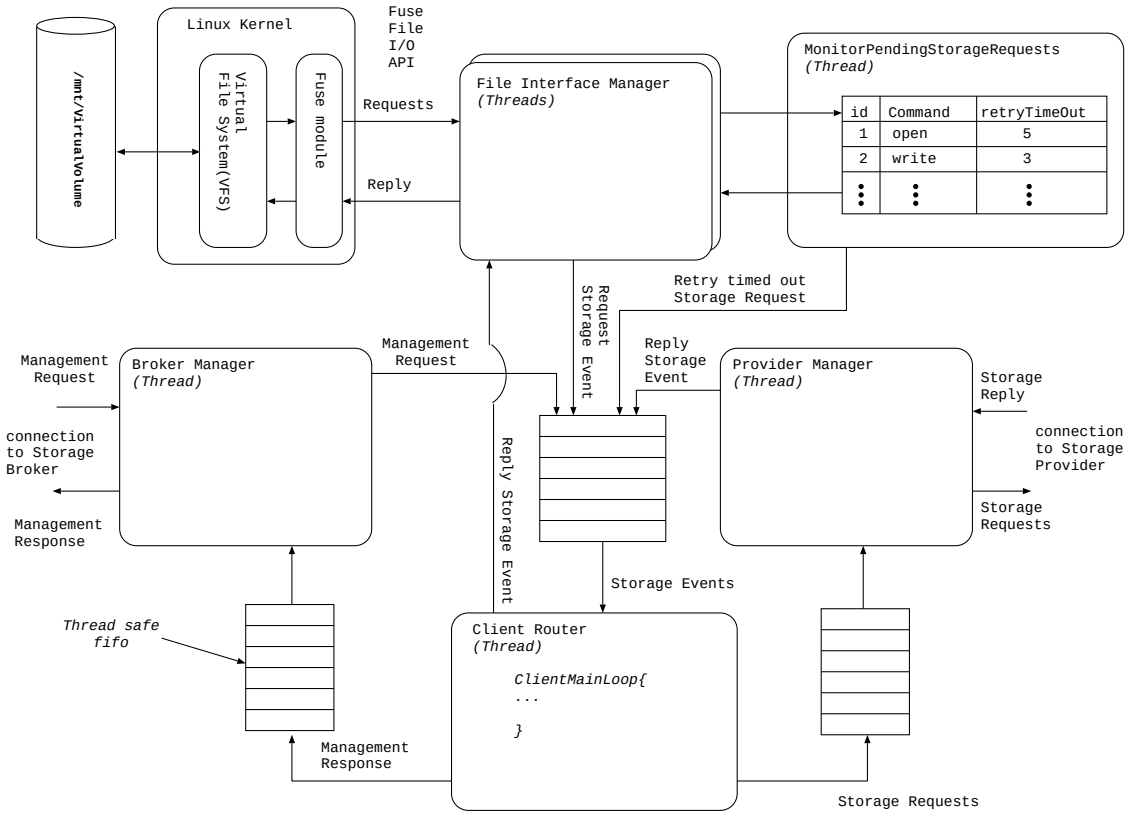

Figure 3.9: Storage Client: threading and message passing

with the Storage Broker and Primary Storage Provider.

3.5 Storage Broker

The Storage Broker was designed to be an institution's gateway to the outside world, responsible for initiating trade negotiations with the Storage Marketplace, authenticating connections from external Storage Clients and monitoring internal storage services. Within the institution, the Storage Broker accepts connections from (i) Storage Providers which report status information, (ii) Storage Clients wishing to access storage in-house and (iii) administrators wishing to configure storage services. In the following three sections, we discuss the architecture, object oriented design and data modelling used in the development of the Storage Broker.

Algorithm 2 Client Router Main Loop

```
1: Input: Incoming storage events on thread safe fifo ISE
2: Output: Provider Storage thread safe fifo PS,
   Reply storage event thread safe fifo RSE
3: PSR ← {∅} // a set used to keep track of pending storage requests
4: for all StorageEvent ∈ ISE do
5:   if StorageEvent is of type file IO then
6:     if StorageEvent is a reqeust then
7:       if isStorageProviderConnected then
8:         PS ← PS ∪ StorageEvent
9:       else if isStorageBrokerConnected then
10:        sendRequestForProviderToBroker()
11:      else
12:        connectToBroker()
13:      end if
14:    else if StorageEvent is a reply then
15:      OSR ← {∅} // Original storage request
16:      OSR ←findTheOrigRequestThisReplyIsFor(PSR, StorageEvent)
17:      if OSR ≠ ∅ then
18:        PSR ← PSR \ OSR // remove from pending list
19:        wakeUpBlockingFuseThread(StorageEvent) // reply passed to File
20:                                               // Interface Manager
21:      end if
22:    end if
23:  else if StorageEvent is of type management request then
24:    if StorageEvent is of type Storage Provider connected then
25:      isStorageProviderConnected=true
26:    else if StorageEvent is of type Storage Broker connected then
27:      isStorageBrokerConnected=true
28:    end if
29:  end if
30: end for
```

3.5.1 Architecture

The Storage Broker architecture consists of the following six main components (Figure 3.10): Command Console Manager, Storage Trader, Client Manager, Storage Event Router, Provider Manager and Storage Broker Database Manager.

1. **Command Console Manager:** The Command Console Manager enables the institution to configure the Storage Broker to meet their storage requirements. The administrator is permitted to create user accounts for internal and external users. There are two ways internal users can create available storage entries,

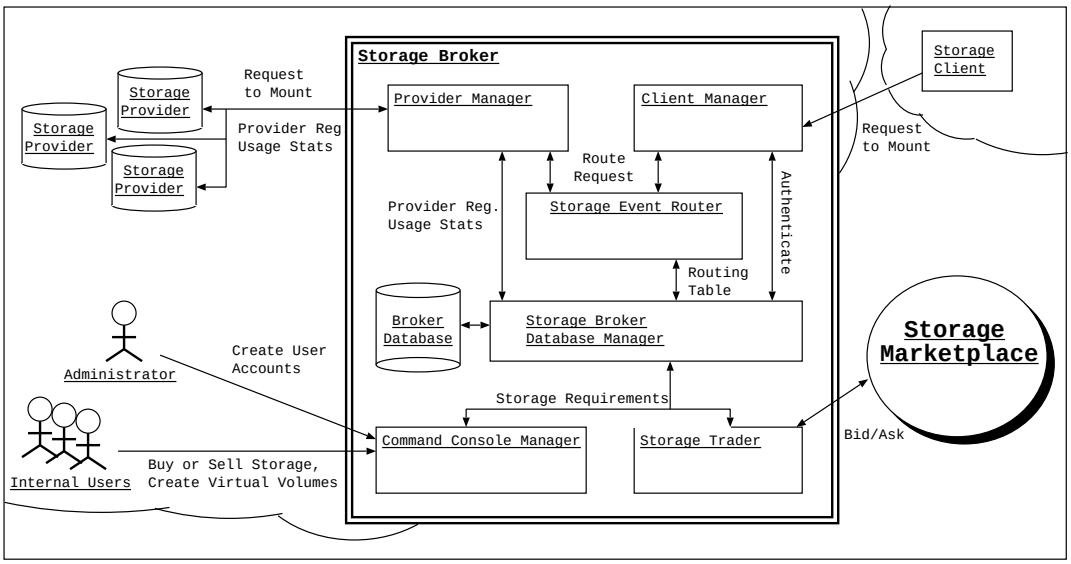

Figure 3.10: Storage Broker: component diagram

(i) by registering Storage Providers with available store, or (ii) by submitting a request to purchase storage external to the institution. An internal user is able to configure virtual volumes which utilise their available store. These Virtual Volumes can then be flagged for internal use or be sold to external institutions. External users of the Command Console Manager are restricted to viewing their purchased Virtual Volume information. The Command Console Manager uses the Storage Broker Database Manager to store storage requirements and user account changes.

2. **Storage Trader:** The Storage Trader is responsible for establishing communications with the Storage Marketplace and relaying storage requirements in the form of bids and asks. It achieves this by querying the Storage Broker Database Manager for either Virtual Volumes that have been flagged for sale or Available Store entries flagged to be purchased, it then advertises these trades to the Storage Marketplace.

   The Storage Trader has much potential for future research and development, as we envisage this component to play a key role in providing autonomic storage. In the future, the Storage Trader could be designed to trade storage services autonomously, rather than waiting for direct user input for trades. Its trading

behaviour may be influenced by volume usage patterns or market status.

3. **Client Manager:** The Client Manager component accepts connections from Storage Clients wishing to access the storage services (Section 3.2.6: Step 1). Upon authenticating the Storage Client, the Client Manager submits the Client's request to mount to the Storage Event Router.

4. **Storage Event Router:** The Storage Event Router is responsible for building a routing table mapping virtual volumes to Storage Providers. When the Client Manager submits a request to mount, the Storage Event Router selects a Primary Provider to service the virtual volume and relays the request to mount to the Provider Manager component. If there are multiple Providers capable of servicing the virtual volume, the first Provider is selected from an unordered list. The process of selecting the Primary Provider has much potential for development as the selection process could take into account load sharing or biasing selection to reliable providers.

5. **Provider Manager:** The Provider Manager component accepts connections from Storage Providers allowing them to register and report their available capacity. A Storage Provider registers by reporting the id of the user it belongs to. Each time a Storage Provider registers, the Provider Manager updates the database with the information provided. When a Provider Manager receives a request to mount from the Storage Event Router, it relays it to the Storage Provider (Section 3.2.6: Step 2), if this is unsuccessful it will need to notify the Storage Event Router.

6. **Storage Broker Database Manager:** Is responsible for servicing queries from each of the components discussed. The Storage Broker Database Manager ensures persistence in the event the Storage Broker is restarted. The Storage Broker Database Manager stores information on virtual volume configurations, user accounts and available storage.

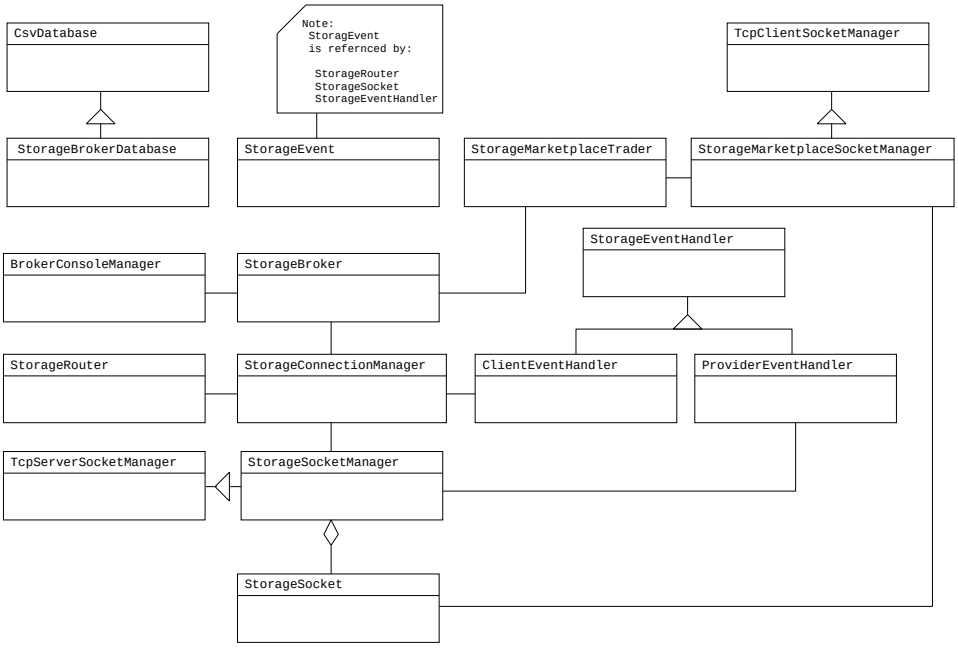

Figure 3.11: Storage Broker: UML class diagram

3.5.2 Object Oriented Design

The Storage Broker was designed using an using an Object Oriented (OO) methodology. This section details the Storage Broker's class diagram (Figure 3.11) and where possible traces back to the architecture:

1. **StorageBroker:** The StorageBroker class contains the main method which is responsible for starting up all the main threads in the system. These include the StorageMarketplaceTrader, StorageConnectionManager and the BrokerConsoleManager.

2. **StorageMarketplaceTrader, StorageMarketplaceSocketManager, TcpClientSocketManager:** The StorageMarketplaceTrader class traces back to the StorageTrader in our architecture and is responsible for initiating trades with the Storage Marketplace. The StorageMarketplaceSocketManager inherits the TcpClientSocketManager to provide the TCP communications used by the StorageMarketplaceTrader to communicate with the Storage Marketplace. The StorageMarketplaceTrader queries the StorageBrokerDatabase to determine what storage should be purchased or sold. If there is any storage

to be traded, this information is relayed to the StorageMarketplace. Upon allocating trades, the Storage Marketplace notifies the StorageMarketplaceTrader of the outcome.

3. **StorageBrokerDatabase, CsvDatabase:** The StorageBrokerDatabase inherits from the CsvDatabase class. The CsvDatabase class provides a layer of abstraction, hiding how the data is stored from other classes. The CsvDatabase class imitates database functionality, but stores table information in a csv file format. This layer of abstraction allows for future integration to production based databases e.g. Oracle or MySQL.

4. **StorageConnectionManager,StorageRouter,StorageSocketManager, TcpServerSocketManager and StorageSocket:** These classes are responsible for handling communications with the Storage Provider and Storage Client components. The StorageConnectionManager handles multiple Storage Providers and Storage Clients connecting concurrently. The StorageConnectionManager manages multiple open sockets by using the StorageSocketManager and uses the StorageRouter to route Storage Clients' requests to mount volume to the correct Storage Provider. TcpServerSocketManager initialises TCP listener ports and accepts connections by notifying the StorageSocketManager with StorageSockets. Much of the functionality provided by these classes traces to the Storage Event Router component in the architecture.

5. **StorageEventHandler, ClientEventHandler, ProviderEventHandler:** These classes are responsible for processing Storage Events from both the Storage Provider and Storage Broker. These classes trace back to the Storage Client Manager and Provider Manager components discussed in the architecture section.

6. **BrokerConsoleManager:** Accepts TCP connections from administrators wishing to configure the Storage Broker. Administrators may use a telnet client to connect to the BrokerConsoleManager which will present them with a command prompt. This class traces to the Command Console Manager

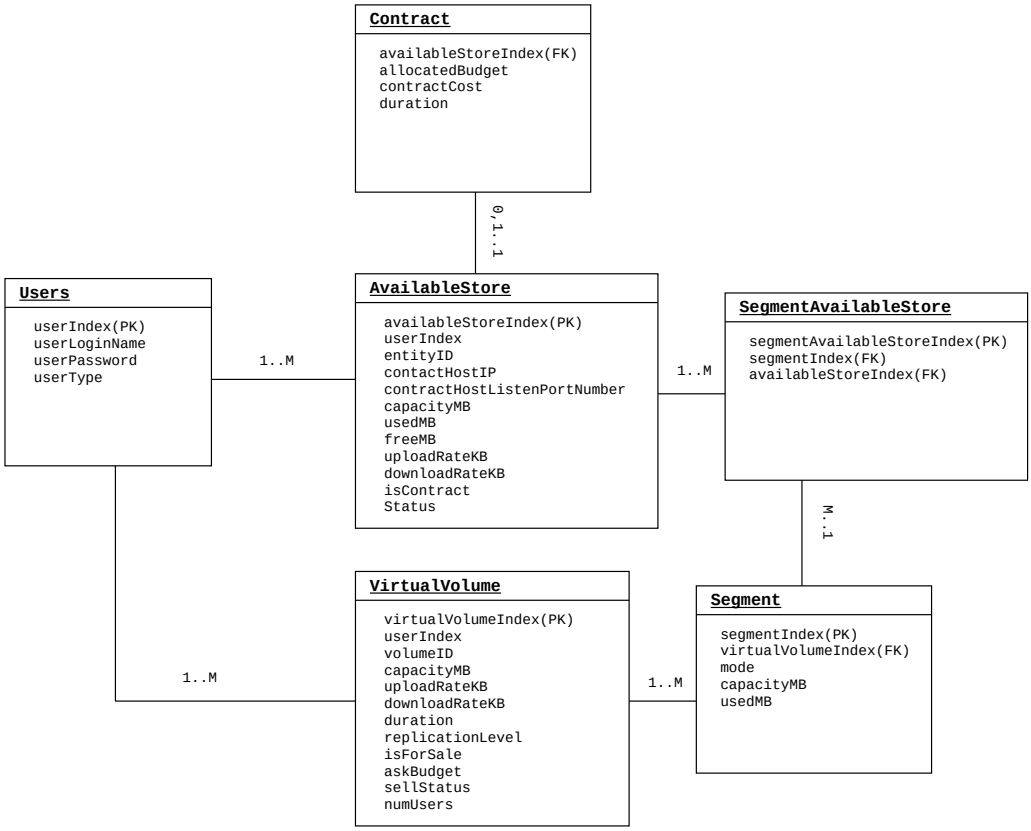

Figure 3.12: Storage Broker: entity relationship diagram

discussed in the architecture section.

3.5.3 Data Modelling

A relational approach was used for the Storage Broker's data modelling. The tables, fields and cardinality is illustrated in our Entity Relationship Diagram (Figure 3.12). We shall describe each of the tables.

1. **Users:** This table stores user account details, ensures that only registered users are able to access the Storage Broker. Users can be assigned many AvailableStore and VirtualVolume entries.

2. **AvailableStore, Contract:** The AvailableStore table is responsible for storing information about the available storage the Storage Broker can utilise. There are two main types of available store, (i) contract: purchased from an external institution or (ii) local: storage provider within the institution. If

an AvailableStore record is a contract, then, contract details are stored in the Contract table.

3. **VirtualVolume, Segment, SegmentAvailableStore:** A VirtualVolume can be stored across many segments, with each segment able to be referenced back to an AvailableStore record via the SegmentAvailableStore table. When a VirtualVolume is created it is assigned a single Segment. Based on the VirtualVolume's replicationLevel (N), the Segment will reference (N) AvailableStore entries via the SegmentAvailableStore table. We introduce the concept of segments to allow the future possibility of Virtual Volumes growing and shrinking as Segments are added and removed.

A field by field description of each table can be found in Appendix A.

3.6 Storage Marketplace

The Storage Marketplace is responsible for accepting bids and asks from Storage Brokers and employing a market model to allocate trades and notify Storage Brokers of the results. In the following two sections, we discuss the architecture and object oriented design used to develop the Storage Marketplace.

3.6.1 Architecture

The Storage Marketplace architecture consists of the following four main components (Figure 3.13) the Market Manager, Storage Broker Manager, Command Console Manager and the Storage Marketplace Database Manager.

1. **Command Console Manager:** The administrator of the Storage Marketplace may use the Command Console Manager to query the database to analyse the trading process.

2. **Market Manager:** The Market Manager is responsible for executing a market model, it periodically queries the Storage Marketplace Database Manager for

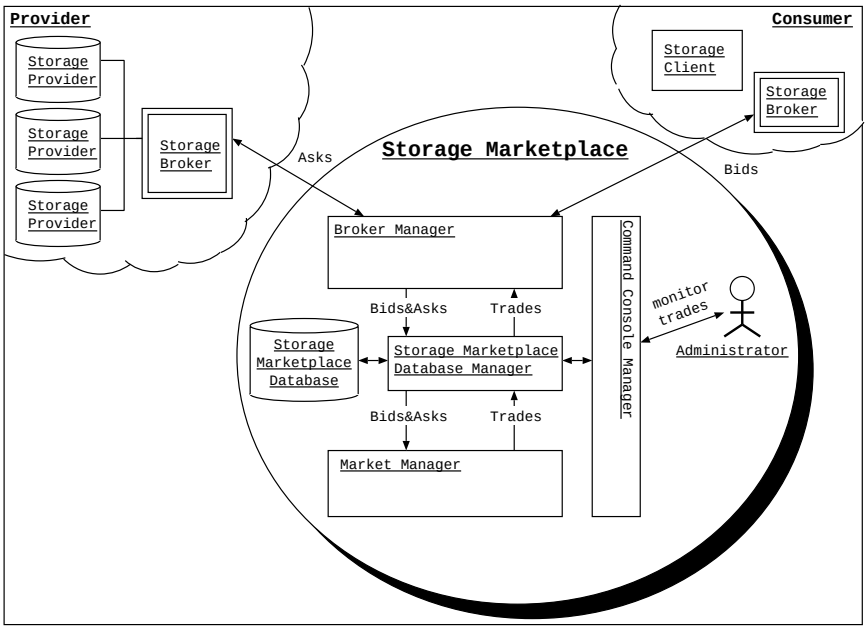

Figure 3.13: Storage Marketplace: component diagram

bids and asks and allocates trades. The trade allocation results are written back to the Storage Marketplace Database Manager.

3. **Broker Manager:** The Broker Manager accepts connections from Storage Brokers, allowing them to submit bids and asks. When a Storage Broker first connects, it is assigned a Broker ID which it uses for all future connections. Bids and asks accepted by the Storage Broker are forwarded to the Storage Marketplace Database Manager to be stored. The Broker Manager also queries the Storage Marketplace Database Manager for trades which have been cleared and relays the results to the respective Storage Brokers.

4. **Storage Marketplace Database Manager:** Is responsible for servicing queries from each of the components discussed. The Storage Marketplace Database Manager ensures persistence in the event the Storage Marketplace is restarted. The Storage Marketplace Database Manager stores bids, asks and allocated trades.

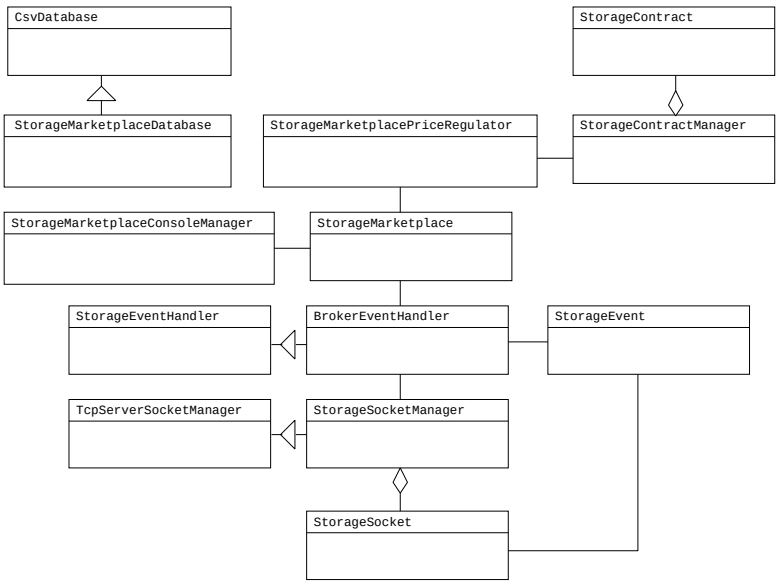

Figure 3.14: Storage Marketplace: UML class diagram

3.6.2 Object Oriented Design

Like the Storage Broker, the Storage Marketplace was designed using an Object Oriented (OO) methodology. This section details the Storage Marketplace's class diagram (Figure 3.14) and where possible traces back to the architecture:

1. **StorageMarketplace:** The StorageMarketplace class contains the main method which is responsible for starting up all the main threads in the system. These include the StorageMarketplacePriceRegulator, BrokerEventHandler and StorageMarketplaceConsoleManager.

2. **StorageMarketplacePriceRegulator, StorageContractManager, StorageContract:** The StorageMarketplacePriceRegulator instantiates four instances of the StorageContractManager, one for every type of Storage Contract: (i) Available Storage Contracts (asks), (ii) Requested Storage Contracts (bids), (iii) Negotiated Storage Contracts (successfully allocated) and (iv) Unfeasible Storage Contracts (unsuccessfully allocated). The StorageMarketplacePriceRegulator determines the available and requested storage contracts by querying the StorageMarketplaceDatabase. Periodically, the StorageMarketplacePriceRegulator executes a clearing algorithm across the Available and Requested storage contracts to determine the Negotiated and

Unfeasible Storage Contracts. The results of the clearing algorithm are also submitted to the StorageMarketplaceDatabase. Much of the functionality provided by these classes traces back to the Market Manager component in the architecture.

3. **StorageMarketplaceDatabase, CsvDatabase:** The StorageMarketplaceDatabase inherits from the CsvDatabase. The CsvDatabase has been reused from the Storage Broker, providing a layer of abstraction, hiding how the data is stored from other classes. The StorageMarketplaceDatabase manages the following two tables: (i) Broker - A registry of all Storage Brokers (ii) Storage Contract - Stores information on contracts currently being processed.

4. **BrokerEventHandler,StorageEventHandler, StorageSocketManager, StorageSocket,StorageEvent,TcpServerSocketManager:** These classes are responsible for handling communications with Storage Brokers. All these classes with the exception of the BrokerEventHandler, have been reused from the Storage Broker. The StorageSocketManager manages multiple connections from Storage Brokers. The BrokerEventHandler receives storage events (containing bids and asks) from the StorageSocket, and upon processing, registers the bids and asks with the StorageMarketplaceDatabase.

5. **StorageMarketplaceConsoleManager:** Accepts TCP connections from users wishing to observe the Storage Marketplace's operation. Users may access the Storage Marketplace by using a telnet client to connect to the StorageMarketplaceConsoleManager, which will present them with a command prompt. This class traces to the Command Console Manager discussed in the architecture.

3.7 Implementation

The Storage Provider and Storage Client components have been written in C. A quick prototype written in Java proved troublesome on two fronts, JNI bindings which interfaced with FUSE were unstable and the poor performance mirrored

experiences found by OceanStore [109]. The Storage Client relies on the FUSE [56] kernel module to provide a local mount point of the storage volume. The decision to use FUSE was based on its ability to:

1. Provide a simple and clean file-system API.

2. Code developed runs in user space which has the benefit of protecting users and developers from buggy code having access to kernel address space, bugs which would otherwise result in the system to crash.

3. Its potential to integrate with EncFS [48], which would allow all file operations to be encrypted on the fly before leaving the Storage Client.

Both the Storage Client and Storage Provider are dependant on file system I/O calls, unix socketing and threading libraries. The network protocol used for communication amongst all components is done via TCP/IP sockets. Messages passed between threads and over sockets are based upon storage event protocol (Appendix B).

To increase speed of development, the Storage Broker and Storage Marketplace were written in Java. Interactions between the Broker, Provider and Client have been implemented and tested. We have been able to successfully mount a replicated storage volume utilising scavenged storage made available by Providers. However, communication between the Storage Marketplace and Storage Broker is not complete. Whilst clearing algorithms proposed in later chapters have been developed and incorporated into the Storage Marketplace, bids and asks are read from file, rather then being submitted by a remote broker.

3.8 Evaluation

In this section we evaluate the file system performance provided by the Storage Exchange platform, using various block sizes.

3.8.1 Experiment Setup

The benchmark we provide involved the Storage Broker, Storage Provider and Storage Client components. The Virtual Volume was configured to replicate twice, across two Storage Providers. Our configuration consisted of three machines, two - (Athlon 900, 256MB RAM, 80Gb HD, Debian 3.0) were the two Storage Providers and one - (P4 1.7Ghz, 1GB RAM, 120GB HD, Debian 3.0) was the Storage Client and Broker. Our benchmark consisted of copying a 5MB file from the virtual volume and repeating this operation for different sizes of read block (configured within FUSE). We varied the read block size by continually doubling it from 4KB, to 128KB. All machines had 100Mbit network cards and were connected to a single 100MB 8-port switch.

3.8.2 Benchmark

We observed the read performance improved in proportion to the block size (Figure 3.15, Table 3.1). With the exception of when block size was increased from 8KB to 16KB, there was a 404% increase in performance. The reason for a 404% increase is difficult to explain, especially as all the other increases in performance are in proportion to block size. The proportionate increases in performance with respect to block size is due to (i) the fact that storage requests are issued synchronously and (ii) network latency. The Storage Client issues a read storage request equal to the maximum set block size and waits for the reply before issuing the next read request. A small block size requires a higher number of requests be issued to read the 5MB file than if a larger block size was used. Even though the block size is small, the cost of latency ensures that it takes the same time for a round trip of a 8k block as a 128k block. Thus explains the proportionate increase in performance in relation to the block size. A 100Mbit LAN is capable of $\approx 10 MBytes/Sec$, in our benchmark only 31% of network capacity was used, highlighting the limitation of the synchronous approach.

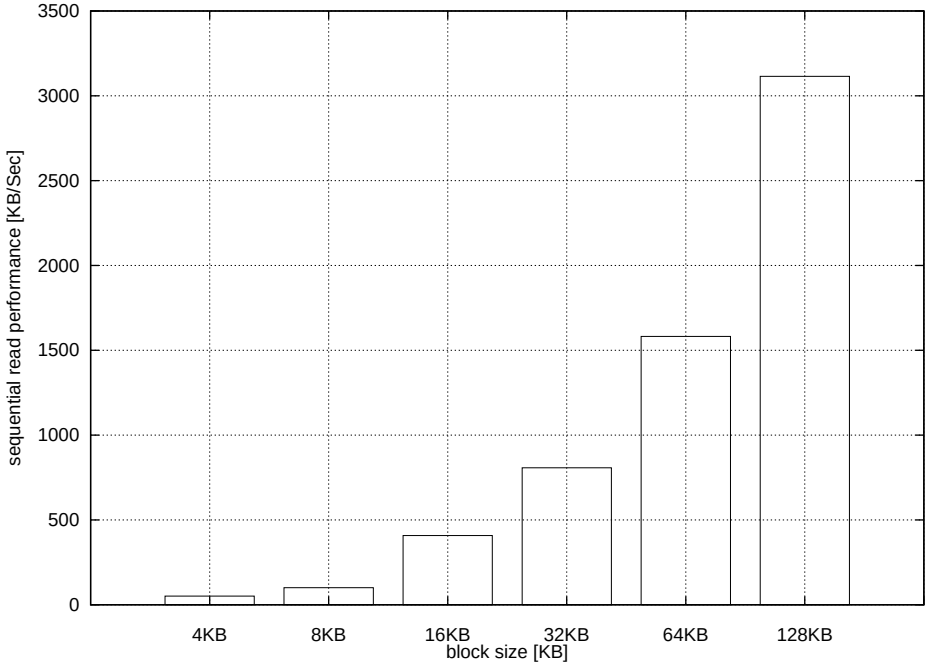

Figure 3.15: Storage Exchange: sequential read performance - varying block size

block size (KB)	***read performance*** (KB/Sec)
4	51
8	101
16	408
32	807
64	1582
128	3115

Table 3.1: Storage Exchange: sequential read performance - varying block size

3.9 Discussion and Summary

In this chapter we proposed a unique global trading platform for distributed storage services. The Storage Exchange allows institutions to share and exchange storage services across global and administrative boundaries.

1. **Interface:** The storage client provides a filesystem like interface and therefore existing applications can utilise storage services without being modified.

2. **Architecture:** The Storage Exchange adopts a centralised architecture, which follows a hierarchical pattern with the Storage Marketplace component at the top followed by the Broker and finally the Client and Provider. Whilst the

Storage Marketplace is a central component and admittedly poses a scalability and reliability bottleneck, it is solely responsible for clearing trades. Hence, if the Storage Marketplace were to become unavailable it will not affect the operation of storage contracts, institutions will continue to be able to mount and access storage services. If an institution's Storage Broker were to fail, volumes already mounted would continue to be serviced by the respective providers, although requests to mount new volumes would fail. Whilst centralised architectures do pose limits, if carefully designed a centralised architecture can be made to scale extremely well [50], the GFS [57] is an example.

3. **Consistency:** To simplify consistency, only a synchronous mode of operation is supported leaving issues of consistency to be resolved by the provider's filesystem. For volumes with multiple replicas, even with synchronous operation, it is still possible for replicas to become inconsistent. To limit the inconsistency, the Storage Provider restricts access if any replica is unavailable, but if a file operation were to succeed on one replica and fail on another there is no capability to rollback changes. One way to overcome this dilemma would be to employ a leasing protocol, allowing changes which fail on one replica to be rolled back on the other.

4. **Performance:** Due to only supporting a synchronous mode of operation, performance is well below network capacity and much slower than distributed file systems like NFS.

At the centre of the Storage Exchange platform is the market model responsible for allocating trades based upon provider's and consumer's storage requests. The process of selecting and applying a suitable market model forms the basis of our next chapter.

Chapter 4

STORAGE EXCHANGE CLEARING ALGORITHMS

The aim of this chapter is to find a suitable market model for the Storage Exchange platform proposed in the previous chapter. This chapter begins by comparing auctions with other market models and continues by discussing One Sided Auctions and Double Auctions. For each auction, we outline the trading process involved, applications in practice, and adopt a distributed systems perspective when discussing implications on architecture, communication overhead and clearing complexity. We provide a summary of auction market models and relate it to distributed storage services. We identify the Double Auction approach to best suit the requirements of the Storage Exchange, despite its practical application being limited to clearing trades where demand is divisible. Clearing trades in a Double Auction, where demand is indivisible, is classified as an NP-hard problem [79] and thus computationally intractable. To overcome this limitation we propose and evaluate four different clearing algorithms with polynomial complexity. We conclude by summarising our results and discuss various trade-offs.

4.1 Auctions

Auctions have proved to be an efficient and flexible market mechanism which quickly converges to a competitive equilibrium [145, 147, 55] under a variety of conditions[1].

Other market mechanisms such as bartering and commodity markets have also proved to be very applicable in practice and within computer systems. The bartering model has been successfully applied by SAV [25] who have found that "A barter system is simpler and more appropriate for an autonomous, peer-to-peer network

[1]Competitive Equilibrium (CE) [55]: A set of prices which equate the demands of utility-maximizing consumers to the supplies of profit maximizing firms. The intersection point of demand and supply curves.

than a system that requires some central entity to control the money supply." Whilst the barter system is ideal for a cooperative environment where entities possess a *double coincidence of wants* [110], it does not accommodate participants which seek to only purchase or sell services.

In a commodity market model, the provider of the services sets the price. The price may be fixed or variable based on supply and demand. Consumers choosing to use a provider service pay at the advertised rate for the amount they use. Commodity market models are widely used, setting prices for electricity, gas, phone and Internet services. The application of fixed or flat rate pricing results have been successfully applied to High Performance Computing (HPC) environments and have shown to scale well whilst maintaining a simple scheduling algorithm [23]. Although a fixed pricing scheme has its limitations, there are no incentives for consumers to limit their usage of the service and thus the allocation strategies suffer from service congestion problems [92, 52]. To overcome congestion problems associated with a fixed price model, providers may employ a congestion based pricing scheme [70]. Unfortunately even though this may overcome some shortfalls in a fixed price model, it still results in allocations which are economically inefficient and unpredictable [23].

In comparison with bartering and commodity market models, auctions provide the necessary flexibility and market efficiency we seek in the Storage Exchange. We now investigate and compare various auction mechanisms.

4.2 One Sided Auctions

An auction is deemed one-sided if only bids or asks are accepted during the life of the auction [55]. There are four basic types of auctions which fall into the one-sided category [91]: English Auction, Dutch Auction, First Price Sealed Bid and Vickrey. Our discussion of each of the auctions inherits the same assumptions presented in [91], these include:

1. Bidders are risk neutral.

2. Bidders make a private independent value of the good.

3. Bidders are symmetric.

4. Payment is a function of the bids alone.

A comparison of one-sided auctions [91] shows that all four auctions yield, on average, the same revenue to the seller. One-sided auctions result in allocations which are Pareto efficient [122][2]. As bidding strategies employed in First Price Sealed Bid and Dutch Auctions require speculation about other bidders, they do not have a dominant strategy. Bidders waste effort on counterspeculation, thus are less efficient than auctions with dominant strategies (English Auctions and Vickrey) [122].

4.2.1 English Auction

An English Auction [91] is an *outcry* ascending auction, where the auctioneer initiates the auction by advertising the good that is up for sale, consumers then participate by submitting bids in an ascending fashion. The auction concludes when the bidding stops and the consumer with the highest bid is declared the winner. There are a few conditions which may be applied to the way English Auctions are executed, these include:

1. **Reserve Price:** An auctioneer may set a reserve price which is unknown to bidders. If the auction concludes without exceeding the reserve, the good remains unsold.

2. **Time Duration:** An auctioneer may set a time limit for how long they are willing to accept bids before concluding the auction.

3. **Incrementation:** An auctioneer may set a minimum increment by which participating consumers need to increment their bid.

4. **Limiting Bids:** An auctioneer may limit the number of bids a consumer is able to submit.

[2]A market model is deemed Pareto efficient if no entity can be made better off without making some other entity worse off.

English Auctions are widely used in practice and are ideal for situations where the seller is uncertain of the value of their goods, or the nature of the goods are unique. An interesting behavioural observation shows the excitement generated by the *outcry* nature of the English Auction results in bidders bidding higher than compared to the rational auctioned good value; hence the winner is left with a good they paid too much for, and suffer from what is deemed as *the winner curse* [93, 138]. The dominant strategy employed when participating in an English Auction is to bid a small amount more than the current highest bid and stop when the private value price is reached.

Communication Overhead

The number of messages that need to be relayed in an English Auction are relatively high, [5] shows an exponential relationship in the number of messages which need to relayed as the number of resources increase. Compared with First Price Sealed Bid, Dutch Auctions and Double Auctions, English Auctions are shown to require the the highest rate of messages [5]. There are a number of reasons why this is so: (i) an English Auction follows an *outcry* method of communication, in a network this translates to broadcasting messages amongst participants, (ii) if no limit is enforced on the number of bids submitted, bidders may potentially submit numerous bids adding to the number of messages relayed in an auction. From our analysis we see the number of messages exchanged per auction (Figure 4.1) to follow:

$$M = C_n(B + 1) + 1 \tag{4.1}$$

where:

M : is the number of messages relayed per auction.
C_n : is the number of participating consumers.
B : is the total number of bids placed by all consumers.

We observe the number of messages exchanged in the process of an English Auction has the potential to be high. To illustrate, if each participant C_n is limited to a single bid ($B = C_n$), then the total number of messages transferred during

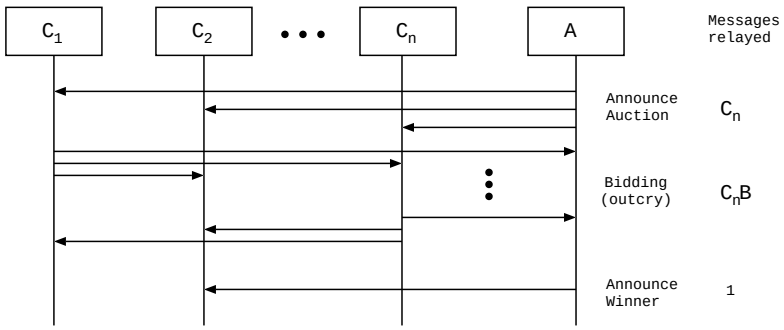

Figure 4.1: English Auction: messages relayed

that auction would be polynomial ($C_n^2 + C_n + 1$), with respect to the number of consumers participating. Such a high communication overhead would pose a limit on scalability.

An interesting bidding behaviour observed on Ebay is last-minute bidding [115] otherwise known as *sniping*, where a wave of consumers rush to submit bids as the the auction is set to conclude. This behaviour has the potential to congest the node hosting the auction, as a result it is not uncommon for consumers to find that they are unable to bid in the closing moments of an auction. Research [89] into discouraging *sniping* proposes introducing incentives for consumers to bid early and avoid the last-minute rush.

4.2.2 First Price Sealed Bid

A First Price Sealed Bid Auction [91] involves the auctioneer initiating the auction by advertising the good that is up for sale, consumers than participate by submitting a single *sealed* bid, unknown to other consumers, the consumer with the highest bid is the winner. The First Price Sealed Bid auction is similar to an English Auction. Whilst in an English Auction, bidders have the ability to revise their bids based on rivals bids, in a First Price Sealed Bid, bidders may only submit one sealed bid.

In practice, First Price Sealed Bids are frequently used by governments when they advertise contracts via a Request for Tender (RFT). Firms than submit bids and the Government, by law [91], chooses the lowest qualified bidder.

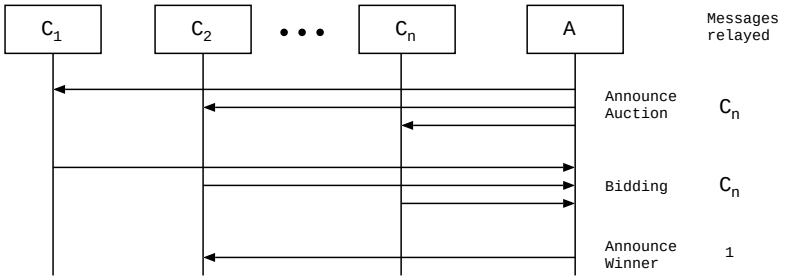

Figure 4.2: First Price Sealed Bid Auction: messages relayed

Communication Overhead

Simulations show that First Price Sealed Bid carries less of a communication burden than an English Auction [5]. As bidders are limited to a single bid and because the bids are sealed, the need to broadcast current highest bids to other participant is not required. We can see the possible number of messages exchanged per auction (Figure 4.2) to follow this relationship:

$$M = 2C_n + 1 \tag{4.2}$$

From this, we can deduce the number of messages exchanged in a First Price Sealed Bid auction to follow a linear relationship. This supports simulation results from [5], showing this auction model to have far less communication overhead than an English Auction.

4.2.3 Vickrey

Vickrey [145] auctions are also referred to as Second Price Sealed Bid auctions. Participating bidders submit a single sealed bid, the bidder with the highest bid wins but at the price of the second highest bid. A bidders dominant strategy in a private value auction is to bid his true valuation. Whilst Vickrey auctions poses nice theoretical properties, they are seldom used in practice [117, 116].

Communication Overhead

Identical to First Price Sealed Bid.

4.2.4 Dutch Auction

In a Dutch Auction [91], the auctioneer begins the auction by announcing a high price and slowly lowers the price until a bidder accepts the current price. Hence, Dutch Auctions are also referred to as descending auctions. Dutch Auctions have the potential to achieve a higher price for the auctioneer than English Auctions [91]. In an English Auction the bidder increases the item price by submitting small increments over previous bids, potentially resulting in the winner paying well below their true valuation. Comparing this to a Dutch Auction, a bidder that really wants the item cannot afford to wait too long and will bid at or near his true valuation.

Dutch Auctions are used in practice in the Netherlands to auction produce and flowers and both England and Israel use them to sell fish. Another not so obvious application of Dutch Auctions can be seen in use by a store in Boston (Filene's) where each item on sale has a price tag with a date attached. The longer the item remains unsold the more discounted it becomes. Therefore when a customer pays at the register, the final price of the item is determined by subtracting the initial with a discount based on how long ago the item was tagged.

Communication Overhead

Simulations show that Dutch Auctions result in relatively high communication overhead, higher than sealed one-sided auctions although less than an English Auction [5]. As the auctioneer is required to broadcast D descending prices to all participating bidders C_n, we can see (Figure 4.3) this has the potential to require a high number of messages. Simulations [5] show that the typical number of messages relayed by Dutch Auctions to follow a linear-polynomial relationship, which coincides with our analytical analysis:

$$M = C_n(D + 1) + 1 \tag{4.3}$$

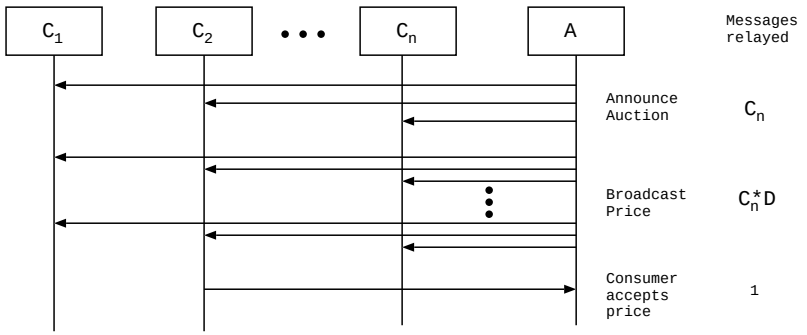

Figure 4.3: Dutch Auction: messages relayed

4.2.5 Summary: One Sided Auctions

In this section we summarise and compare one-sided auctions based upon architecture, clearing complexity and communication overhead.

1. *Architecture:* In practice, one-sided auctions employ a centralised architecture (e.g. Ebay) where auctions are hosted by a central entity where providers and consumers advertise goods for sale and submit bids respectively. Whilst this example demonstrates that one-sided auctions function in a centralised configuration, there is no reason why a more decentralised approach could not be adopted. To illustrate, an auctioneer could host the auction themselves and invite consumers to bid, eliminating the need for a central entity to host the auction. Therefore, one-sided auctions could be employed across a decentralised architecture, like peer-to-peer, where a participating entity could behave as an auctioneer (hosting their own auctions) and as a consumer.

2. *Clearing Complexity:* With the exception of Dutch Auctions where the trade is awarded to the first consumer to accept a price, the process of clearing a trade in one-sided auctions requires a linear scan of bids. In the case of English and First Price Sealed Bid auctions, this means finding the highest bid and for Vickrey, the second highest bid.

3. *Communication Overhead:* In a one-sided sealed auction, there is no advantage to be gained by submitting bids faster than a competitor. Therefore, participants with slow connectivity are not disadvantaged. Also, due to their sealed

nature, these auctions do not require information to be broadcast amongst participants, thus markedly reducing the communication overhead. On the contrary, auctions which are *outcry* based (English/Dutch) require information to be broadcast amongst participants resulting in higher communication overhead then sealed based auctions. These type of auctions also encourage *sniping* and give advantage to consumers with high speed connectivity.

4.3 Double Auction

A Double Auction (DA) [55] market allows both buyers (consumers) and sellers (providers) to submit offers to buy (bid) and sell (ask) respectively. Providers and consumers submit asks and bids simultaneously and hence participate in a double-sided auction. The process of clearing determines the way in which trades are allocated amongst the asks and bids. There are two ways in which clearing may take place, continuously or periodically. Double Auctions cleared continuously are referred to as Continuous Double Auctions (CDA), where bids and asks are submitted in an *outcry* fashion; with compatible bids and asks cleared instantaneously. Double Auctions cleared periodically operate by accepting *sealed* bids and asks, and at the end of the period allocating trades amongst the queued up bids and asks. Double Auctions of this nature are referred to as Clearinghouse (CH), call market or sealed Double Auctions.

Decades of research and experiments [132, 55, 59, 120] show that Double Auctions are an effective and efficient market model, quickly converging towards a Competitive Equilibrium (CE). The CE is the intersection point of true demand and supply curves, yielding allocations which are near 100% efficient. A market that is 100% efficient has no other possible allocation that would result in a larger aggregate benefit to both consumers and providers. Rustichini et. al. [120] provide an analytical study of a sealed Double Auction (k-DA) showing that in any equilibrium the amount by which a trader misreports is $O(1/m)$ and the corresponding inefficiency is $O(1/m^2)$. From an economic stand point, DAs are a sound and efficient market model. The New York Stock Exchange (NYSE) and

Chicago Commodities market both employ a CDA market model. Sealed Double Auctions are used to determine the daily opening price of each stock listed on the NYSE where bids and asks are collected overnight and cleared in the morning. Sealed Double Auctions are also used twice a day to fix copper and gold prices in London.

Architecture

In practice, DAs are executed by a trusted central entity, responsible for accepting bids and asks and clearing trades. The same trusted central entity is required when applying DAs to computer systems. This centralised architecture may pose a scalability and performance bottleneck if not carefully considered. Although there is research [36, 101] which applies a DA approach in a peer-to-peer environment, it is in its early stages of development and the authors acknowledge that many challenges remain to making it a reality. An alternative solution to address centralisation would be to employ a Byzantine agreement protocol [21, 18].

Computational Complexity

In practice, DAs are used with goods which are abstract and divisible. For a stock exchange that would be currency and shares. If DAs are applied to goods with divisible constraints, the process of clearing becomes non trivial requiring a prohibitive amount of computation. A study by Kalagnanam [79] provides an insight into the computational complexity associated with clearing sealed Double Auctions:

1. **Indivisible Demand:** If demand is indivisible, the request in a single bid may only be met using the supply from a single ask. Finding an optimal clearing allocation becomes computationally intractable and requires solving an NP-hard optimisation problem.

2. **Assignment Constraints:** If assignment constraints exist between bids and asks, they can only be cleared optimally in polynomial time using network flow algorithms.

3. **Divisible and No Assignment Constraints:** If demand can be supplied

from many asks and there are no assignment constraints then clearing trades can be done in log-linear time. This scenario is most employed in practice.

Communication Overhead

A study [5] shows that sealed DAs relay a near linear number of messages with respect to the number of resources and have the lowest communication overhead than any one-sided auction. Let us take an analytical approach to derive the number of messages transferred per transaction. Assuming consumers and providers are participating in a sealed DA where only one sealed bid is allowed per period (Δt), we can see that the total number of messages relayed during that period to be (Figure 4.4):

$$M_{total} = \frac{3}{2}(C_n + A_n) \tag{4.4}$$

Although, during one period of a sealed DA, it is possible for multiple transactions to be allocated and therefore we extend our analysis to how many messages are relayed per transaction $M_{perTransaction}$. An acceptable assumption when analysing DAs [55] is to let the number of consumers and auctioneers be equal $C_n = A_n$ and the bids and asks follow a uniform relationship where the supply and demand graph is linear. With these assumptions only half the number of C_n bids can be successful and we find:

$$M_{perTransaction} = \frac{TotalNumberOfMessages}{TotalNumberOfTransactions} = \frac{\frac{3}{2}(C_n + A_n)}{\frac{1}{2}(C_n)} = 6 \tag{4.5}$$

As DAs have the ability to clear multiple transactions instantaneously, the number of messages required per transaction is constant. This is a remarkable property, backing up results from [5] which show DAs to require the least communication.

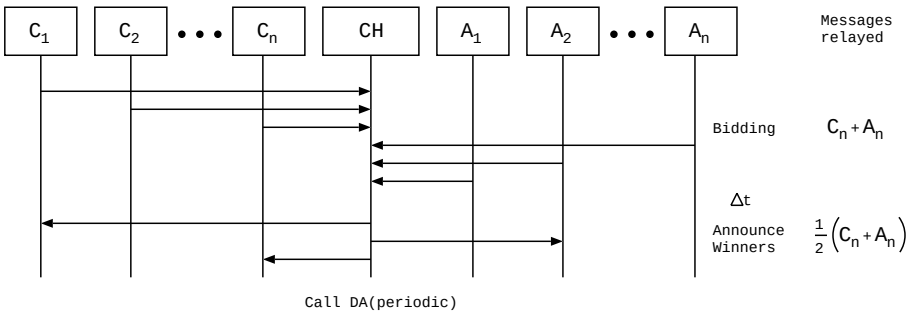

Figure 4.4: Double Auction: messages relayed

4.4 Storage Exchange: A Market Perspective

In this section we discuss market level requirements of our Storage Exchange, covering consumer and provider requirements and compare these with related work discussed in the previous section. We then draw upon our earlier discussions of auction models and evaluate which model best meets the requirements imposed by the Storage Exchange.

The primary goal of the Storage Exchange is to provide institutions with a platform in which they may trade and exchange distributed storage services. In any market there are providers and consumers, in the Storage Exchange providers aim to lease distributed storage services and consumers seek to purchase these services. A key step in choosing a suitable market model and trading storage services is understanding the nature of the goods being traded. The Storage Exchange shall only trade distributed storage services and therefore does not aim to be a universal trading platform for unique individual items. Both consumers and providers have unique service configurations and budgetary constraints and through the use of *storage policies* are able to specify them.

Storage policies incorporate the necessary attributes for providers and consumers to specify their requirements and are essential regardless of market model. Systems such as the one proposed in [38] use storage policies as a way in which to specify high-level Quality of Service (QoS) attributes, hiding low-level error prone administration configurables from the user. Our use of storage policies is similar, encompassing the following attributes:

1. **Storage Policy Attributes (SPA):**

(a) **Capacity:** Storage Capacity (GB).

(b) **Upload Rate:** Rate (KB/sec) of transfer to the volume.

(c) **Download Rate:** Rate (KB/sec) of transfer from the volume.

(d) **Time Frame:** Lifetime/Duration (sec) of storage policy.

Understanding the marketing requirements posed by the Storage Exchange, our discussion shall now re-visit the market models discussed earlier and select a market model which most suites our circumstance.

One-Sided Auctions:

One-sided auctions are ideal for providers selling unique goods with uncertain value. Whilst they exhibit relatively high communication overheads (Table 4.1), one-sided auctions do not require a central entity to oversee their execution, allowing a decentralised architecture to be employed where any peer can host an auction. One-sided auctions require consumers to find an auction which best suites their requirements, for the Storage Exchange, this introduces two inefficiencies, extra communication required to find a suitable auction and as each consumer has unique requirements, it is unlikely they will find an auction to perfectly suit. In one-sided auctions, there may only be one winner leaving all other bidders to search for another auction and thus resulting in further communication overheads. If consumers have time constraints, then meeting these constraints is made difficult in one-sided auctions as consumers are left with participating in auctions which end soon or forced into *buy now* auction. It is likely that in both scenarios, the consumer will pay a higher price.

Double Auction:

As well as being economically sound there are two attractive features of sealed Double auctions which come to our attention, (i) many trades can be cleared instantaneously and (ii) by applying a sealed approach, the need to continuously

[3] Although DAs require an element of centralisation, research into overcoming this exists [36, 101].

Market Model	***Architecture***	***Clearing Complexity***	***Communication Overhead***
English Auction	Decentralised	Linear	Polynomial-unbounded
First Price Sealed Bid		Linear	Linear
Second Price Sealed Bid		Linear	Linear
Dutch Auction		Constant	Linear-polynomial
Double Auction	Centralised[3]	Linear-NP*hard*	Constant-linear

Table 4.1: comparison of auction market models

broadcast current market status is eliminated. Studies comparing Double Auctions [5, 94] with one-sided auction protocols (Dutch, English, Vickrey, First Price Sealed Bid) found that Double Auctions possess the least communication overhead, which is in-line with our analysis showing that under certain circumstances, a constant number of messages are relayed per transaction. Double Auctions do not require consumers to search for auctions which match their requirements, instead both consumers and providers submit their bids and asks and leave the clearing algorithm to allocate trades.

Whilst Double Auctions possess many attractive attributes, (Table 4.1) their conventional application is limited to trading abstract and divisible entities (e.g. currency and shares as found in today's stock exchanges). Upon first glance it would seem the DA market model would be the ideal solution. With distributed storage being a divisible entity, its application to the Storage Exchange should be straight forward. Unfortunately, this statement happens to be naive: (i) it may not be practical to purchase storage for a particular volume across multiple different providers, resulting in demand that is indivisible and (ii) storage policies contain multiple attributes, further complicating the clearing process by introducing assignment constraints.

Kalagnanam's [79] study into Double Auctions shows that if demand is indivisible in auctions, then finding an optimal clearing solution becomes computationally intractable and requires solving an NP-hard optimisation problem. The same study also shows that clearing trades with assignment constraints is a polynomial time process. Clearing challenges aside, the DA market model is an ideal trading

mechanism for the purposes of the Storage Exchange; with comparatively low communication overheads and a symmetric market model allowing both consumers and providers to specify their exact requirements thus eliminating the need to search for matching resources. These attractive properties motivate our investigation and proposal of computationally feasible clearing algorithms allowing the DA to allocate trades where demand is indivisible and goods contain multiple attributes. The research described in the following section proposes polynomial time clearing algorithms.

4.5 Clearing Algorithms

Periodically, the Storage Exchange will match queued up Storage Request Bids (SRBs) with Storage Service Asks (SSAs), the manner in which it does so is determined by the clearing algorithm it employs. We propose and investigate the following clearing algorithms in the context of our Storage Exchange platform:

4.5.1 First Fit

SRBs are allocated to SSAs on a First Fit basis. An SSA is deemed to fit if it has the storage resources required by the SRB and the SSA's cost function returns a price within the SRB's bid amount. SRBs are processed in the order in which they have been queued. This algorithm cannot be applied in practice as its allocation strategy does not take into consideration market surplus, resulting in allocations which are unfair to both consumers and providers. This algorithm serves to provide a *sanity check* of sorts for the algorithms subsequently proposed.

4.5.2 Maximise Surplus

This clearing algorithm aims to maximise the profit of the auction. An SRB is allocated to an SSA which results in the maximum difference between a consumer's bid price and result of the provider's cost function.

4.5.3 Optimise Utilisation

This algorithm focuses on achieving better utilisation by trying to minimize the *left overs* that remain after an SRB is allocated to an SSA. A measure of fit is calculated (Algorithm 3) between an SRB and each SSA. A large measure of fit indicates that the remaining ratios have a large spread amongst each of the *Storage Service Attributes* and therefore would result in an SSA with potentially more waste. Whereas a small population variance would indicate that the remaining *Storage Service Attributes* within the SSA would have less waste. Upon calculating a measure of fit between the considered SRB and each SSA, we allocate it to the SSA which returned the smallest measure of fit. SRBs are processed in the order in which they have been queued.

To illustrate, we provide an example scenario with one SRB and two SSAs (Figure 4.5). From the example we can see that if SRB_1 is allocated to SSA_1 the remaining resources upon allocation would result in SSA_1 having capacity=4, upload=17, download=4, we can see there is much potential for waste as remaining upload is very high at 17 and capacity and download is low at 4. If SRB_1 is allocated to SSA_2 the left overs are more even with capacity=8, upload=7, download=8 and less chance of waste due to one attribute running out and leaving large values remaining and wasted as with allocating SRB_1 to SSA_1. We can see that allocating SRB_1 to SSA_1 would result in a relatively high measure of fit with 0.045 as compared to 0.0038 if SRB_1 were to be allocated to SSA_2. Applying the Optimise Utilisation would result in SRB_1 being allocated to SSA_2 as this has the lowest measure of fit.

Much like the First Fit algorithm, the Optimise Utilisation allocations do not take into account market surplus and thus would result in allocations which are unfair to consumers and providers. This algorithm is the platform for the next algorithm which tries to balance achieving good utilisation and a good auction surplus.

4.5.4 Max-Surplus/Optimise Utilisation

This clearing algorithm (Algorithm 4) incorporates the last two allocation strategies and aims to draw a balance between them. Parameter (k) serves to bias the balance,

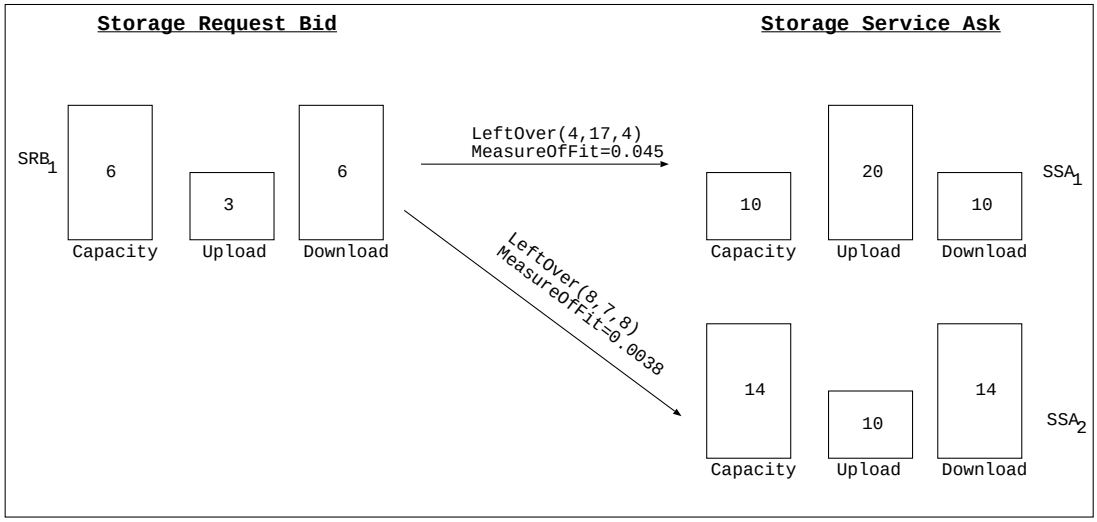

Figure 4.5: Optimise Utilisation Algorithm

Algorithm 3 MeasureOfFit(S,A)

1: **Input**: Storage Request Bid S, Storage Service Ask A
2: **Output**: Measure of Fit F
3: $A = \{a_1, a_2, ..., a_n\}$//*Storage Service Attributes*
4: //belonging to Available Storage Policy
5: $S = \{s_1, s_2, ..., s_n\}$//*Storage Service Attributes* belonging to Storage Request
6: // calculate a remaining ratio for each of Storage Service Attributes
7: $R = \{r_1 = \frac{a_1 - s_1}{a_1}, r_2 = \frac{a_2 - s_2}{a_2}, ..., r_n = \frac{a_n - s_n}{a_n}\}$
8: // calculate the population variance amongst the remaining ratios
9: $F = \frac{1}{n}\sum_{i=1}^{n}(r_i - u_R)^2, where\ u_R = \frac{1}{n}\sum_{i=1}^{n} r_i$

$(0 <= k < 0.5)$ means importance will be given to utilisation, whereas $(0.5 < k <= 1)$ will give importance to achieving a better surplus. Algorithm 4 is applied to every SRB, in the order in which they have been queued.

4.6 Performance and Evaluation

The aim of our experiments is to evaluate each clearing algorithm based upon utilisation and auction surplus. It is important to consider utilisation as this will gauge the efficiency of resource allocation, whilst auction surplus indicates the market efficiency of the algorithm. It is imperative that the clearing algorithm maintains market efficiency as otherwise allocations not only become inefficient but also impractical. The First Fit algorithm is used to confirm that the Maximise

Algorithm 4 Max-Surplus/Optimise Utilisation Algorithm

Input: Storage Request Bid S, Storage Service Asks A, Balance k
Output: Selected Storage Policy P
$F \leftarrow \{\emptyset\}$ // a set to store MeasureOfFit values
$M \leftarrow \{\emptyset\}$ // a set to store Surplus calculations
for all $availableStoragePolicy \in A$ **do**
  if $availableStoragePolicy$ has greater resource attributes than S and S bid price is greater than $availableStoragePolicy$ reserve **then**
    $F \leftarrow F \cup \text{MeasureOfFit}(S, availableStoragePolicy)$
    $M \leftarrow M \cup \text{surplus}(S, availableStoragePolicy)$
  end if
end for
$minSurplus = \min(M)$
$worseFit = \max(F)$
$deltaMeasureFit = worseFit - \min(F)$
$deltaSurplus = \max(M) - minSurplus$
$currentHighScore =$ Large Negative Number
for all $availableStoragePolicy \in A$ **do**
  $ratioBetterFit = \frac{worseFit - MeasureOfFit(S, availableStoragePolicy)}{deltaMeasureFit}$
  $ratioBetterSurplus = \frac{surplus(S, availableStoragePolicy) - minSurplus}{deltaSurplus}$
  $score = (1-k) * ratioBetterFit + k * ratioBetterSurplus$
  if $score > currentHighScore$ **then**
    $currentHighScore = score$
    $P \leftarrow \{availableStoragePolicy\}$ // assign Storage Policy with max score
  end if
end for

Surplus and Optimise Utilisation algorithms actually improve market surplus and utilisation respectively. Finally, Max-Surplus/Optimise Utilisation algorithm is evaluated with the following values of $k = \{0.25, 0.5, 0.75\}$.

Our experiments cover three scenarios. For every scenario each algorithm is executed, allowing us to evaluate how each algorithm performs in different scenarios. Every experiment consists of a single clearing period, where the set of bids and asks processed is equivalent to being queued up over some period of time. Our experiments focus on evaluating the process of clearing at the end of that period. Details of each scenario and the parameters used to generate the data is covered in Section 4.6.1. Section 4.6.2 and Section 4.6.3 present results and discuss their significance.

4.6.1 Experiment Setup

For each scenario, a series of bids (SRBs) and asks (SSAs) are generated by a perl script which complies to the posting protocol otherwise used by consumers and providers. The parameters used to configure the perl script are defined in Table 4.2. Parameters with ranges are assigned with randomly generated numbers within the specified range. The budget assigned to each SRB or SSA is derived from the following linear budget function:

$$BudgetFunction(C, U, D, T) = ((C + U + D)T)pv \tag{4.6}$$

C : Storage Capacity (GB) of volume.
U : Upload Rate (KB/sec).
D : Download Rate (KB/sec).
T : Duration.
pv : percentage variance ($50\% <= pv <= 150\%$).

For each scenario, every clearing algorithm is executed with the same set of bids and asks, which are loaded in the same order in the Storage Exchange. This ensures that for each scenario the clearing algorithm is executed in exactly the same manner.

Parameter	*Description*
SRB	Number of Storage Request Bids
SRC_{range}	Storage Request Capacity range (GB)
SRU_{range}	Storage Request Up Rate range (KB/sec)
SRD_{range}	Storage Request Down Rate range (KB/sec)
$SRDU$	Storage Request Duration (sec)
$SRBB_{budget}$	Storage Request Budget
SSA	Number of Storage Service Asks
SAC_{range}	Storage Ask Capacity range (GB)
SAU_{range}	Storage Ask Up Rate range (KB/sec)
SAD_{range}	Storage Ask Down Rate range (KB/sec)
$SADU$	Storage Ask Duration (sec)
$SSAB_{budget}$	Storage Ask Budget

Table 4.2: experiment parameters

For each scenario we vary the range of the Storage Policy Attributes (SPA) (Table 4.3) for bids $SRB_{SPA} = \{SRC_{range}, SRU_{range}, SRD_{range}\}$ and asks $SSA_{SPA} = \{SAC_{range}, SAU_{range}, SAD_{range}\}$ with the exception of duration, which is kept

constant. The duration is kept constant as otherwise it posed an overbearing constraint on the clearing process, eliminating most of the trades. The three scenarios we cover are used to determine how each algorithm performs when varying SSA_{SPA} and SRB_{SPA}:

1. **Scenario A:** ($SSA_{SPA} \approx SRB_{SPA}$): In this scenario SSA_{SPA} attributes ranges are the same as found in SRB_{SPA} attributes. Therefore, it is most likely that a single ask will serve a single bid, rarely any more. This implies that both provider's and consumer's storage service requirements are approximately equivalent.

2. **Scenario B:** ($SSA_{SPA} > SRB_{SPA}$): In this scenario SSA_{SPA} attributes ranges are ten times greater than SRB_{SPA} and therefore it will be common for multiple bids to be serviced by a single ask. This implies that providers have a larger quantity of storage they wish to sell to consumers which require relatively smaller amounts of storage.

3. **Scenario C:** ($SSA_{SPA} >> SRB_{SPA}$): In this scenario SSA_{SPA} attributes ranges are one hundred times greater than SRB_{SPA} and therefore it will be very common for multiple bids to be serviced by a single ask. Clearly resulting in a large discrepancy between provider's and consumer's storage service requirements, with providers submitting asks with significantly larger quantities of storage than consumers.

Whilst the SPA ranges vary for consumers and providers in Scenario B and C, we ensure that demand and supply for all the scenarios is relatively even, such that the totals for each of the attributes is $\approx \pm 2\%$. Maintaining a balanced supply and demand for all scenarios allows results across each of the scenarios to be compared. To balance the supply and demand, we ensure the ratio between SSAs and SRBs is proportionately opposite to their SPAs (Table 4.3). Note that the simulation does not cover $SSA_{SPA} < SRB_{SPA}$ as all the trades would fail to clear because demand is indivisible, which requires a single SRB to be serviced by a single SSA.

Parameter	Scenario A	Scenario B	Scenario C
SRB	500	500	5000
SRC_{range}	5 - 50	5 - 50	5 - 50
SRU_{range}	5 - 50	5 - 50	5 - 50
SRD_{range}	5 - 50	5 - 50	5 - 50
SSA	500	50	50
SAC_{range}	5 - 50	50 - 500	500 - 5000
SAU_{range}	5 - 50	50 - 500	500 - 5000
SAD_{range}	5 - 50	50 - 500	500 - 5000

Table 4.3: experiment scenarios

4.6.2 Performance Results

This section provides performance benchmarks for each of the scenarios described. The experiment was conducted on (P4 1.7Ghz, 1GB RAM, 120GB HD, Debian 3.0). The results (Table 4.4) show the clearing algorithms to be of polynomial complexity with the number of bids and the number asks being the main influence on processing time taken to allocate trades. The performance results between Scenario A and Scenario B show an anomaly. According to the complexity analysis Scenario A should take ten times longer to compute than Scenario B, however, the performance results indicate similar times for the two. Upon further investigation, we found the reason for this anomaly was due to the fact that Scenario B had a greater rate of successful trades than Scenario A. We found successful trades to require extra computation for allocating storage contracts and calculating left over storage services, thus extending the computation time for Scenario B. When comparing Scenario B and Scenario C, which have similar rates of successfully allocated trades, the computation time increases consistently with the complexity analysis.

4.6.3 Market Results

For each scenario, results have been broken down into four plots which help us evaluate how each algorithm performs with respect to auction surplus and utilisation efficiency. The horizontal axis represents the number of bids (SRBs) that have been processed during that clearing process. The four plots used are detailed below:

[4]The average time taken was calculated across the different runs for (k=0.25,k=0.5,k=0.75)

Clearing Algorithm	**Complexity**	**Scen. A** A=500 B=500	**Scen. B** A=50 B=500	**Scen. C** A=50 B=5000
First Fit	$O_1(BA)$	1129ms	1099ms	10168ms
Maximise Surplus	$O_2(BA)$	1037ms	1031ms	9974ms
Optimise Utilisation	$O_3(BA(C^2))$	2794ms	3528ms	37673ms
Max-Surplus Optimise Utilisation[4]	$O_4(O_2 + O_3 + A)$	10043ms	7652ms	70562ms

Where:
A: the number of asks
B: the number of bids
C: number of attributes in storage policy

Table 4.4: clearing algorithm performance

1. **Auction Surplus**: For each SRB that is allocated to an SSA, the difference between the consumer's bid price and the provider's ask price is the auction surplus. The Auction Surplus plot aggregates this difference as each SRB is being processed. This plot will indicate how each algorithm performs in auction surplus, with Maximise Surplus expected to do well in all scenarios.

2. **Percentage of Ask Budget Met**: Every SSA is assigned a budget constraint. From this budget, the Storage Exchange determines the rate at which it should charge out the service. Therefore, it is possible and likely that the entire SSA will not be utilised and hence the ask budget will not be fully met. This plot shows the percentage of total ask budget met across all the SSAs being processed.

3. **Percentage of Unsold Storage**: Every SSA is assigned a capacity (SAC) representing the amount of storage it has up for lease. This plot shows the percentage of that capacity which remains unsold.

4. **Percentage of Unfeasible Bids**: A bid is deemed to be unfeasible if there does not exist an SSA which can service it for the price requested. This plot will indicate how each algorithm performs in utilisation rates, with Optimise Utilisation expected to perform well.

Auction Surplus and Percentage of Ask Budget met focus on budget aspects whilst the Percentage of Unsold Storage and Percentage of Unfeasible Bids focus on utilisation achieved. As the supply and demand in each scenario is approximately equal, the DA allocation [55, 120] should result in approximately 50% of bids and asks to be successful allocated. Therefore, our experiment results should result in approximately 50% of ask budget met, 50% of unsold storage and 50% of bids deemed to be unfeasible.

Scenario A

In this scenario the generated SRB and SSA service attributes closely match each other. As a result, on average a single ask will only be able to service a single bid. We expect this to be a worst case scenario, as fragmentation will be unavoidable leading to poor utilisation and inefficient allocation. For the Optimise Utilisation algorithm to perform well, asks should be large enough to service multiple bids, allowing it to better fit bids. This scenario effectively nullifies the Optimise Utilisation algorithm and we can see from the percentage plots (Figure 4.7) and table of results (Table 4.5) where there is little difference between each algorithm (±5%) results. At the end of the clearing period, the percentage plots results show that ≈ 40% of ask budget is met, ≈ 55% of storage remains unsold and ≈ 45% bids failed to clear (unfeasible).

Algo.	Scenario A (%)			Scenario B(%)			Scenario C(%)		
	ABM	US	UFB	ABM	US	UFB	ABM	US	UFB
FirstFit	43.3	52.9	45.4	63.0	37.8	36.8	57.4	40.3	41.4
OptUtil	41.0	53.7	46.4	72.6	25.3	23.2	70.4	23.4	24.2
MaxSur	34.9	57.9	48.6	47.1	49.8	49.2	42.3	50.9	51.9
k=0.25	38.7	54.7	48.6	56.1	38.3	37.2	50.0	41.1	41.9
k=0.5	36.5	56.2	50.2	52.8	41.3	39.4	47.5	44.7	45.5
k=0.75	35.3	57.5	51.4	49.9	44.7	44.2	46.2	46.1	47.1

Table 4.5: market allocation results

The auction surplus plot is the only graph where a notable difference amongst the algorithms is made, with Maximise Surplus, k=0.75 and k=0.5 achieving the best auction surplus, followed by k=0.25, Optimise Utilisation and First Fit. In this scenario there is no significant benefit in using Max-Surplus/Optimise Utilisation

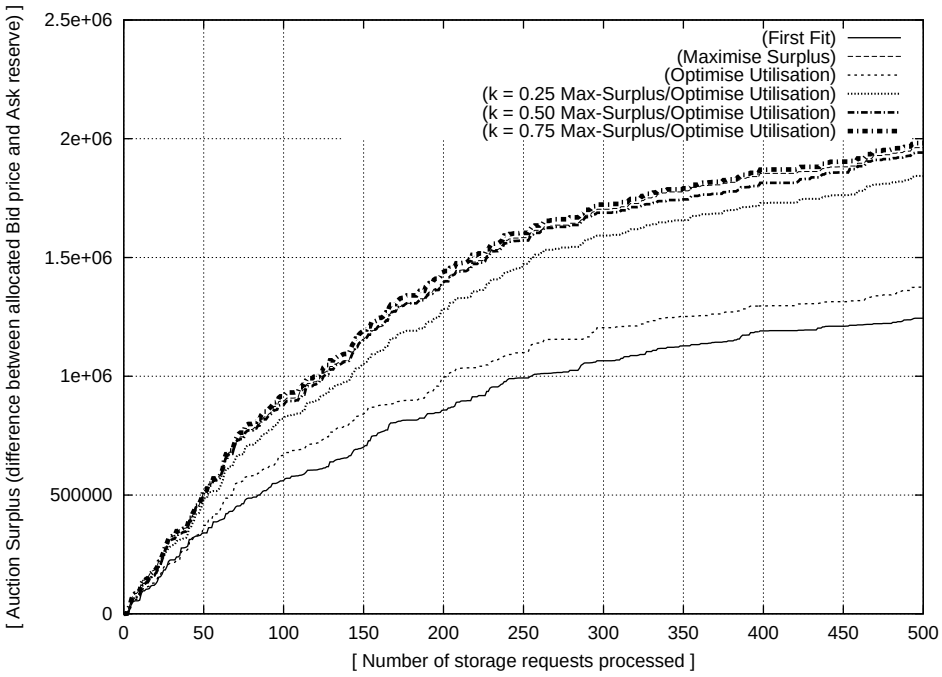

Figure 4.6: Scenario A: auction surplus

and therefore the Maximise Surplus algorithm is the most appropriate approach for this scenario.

Scenario B

In this scenario a single SSA has the potential to serve multiple SRBs and because of this the utilisation algorithm is able to better allocate the bids to asks. The impact is significant (Table 4.5) when compared to Scenario A, with 53.7% of unsold storage in Scenario A dropping to 25.3% and 46.4% of unfeasible bids in Scenario A dropping to 23.2%. We can see that Max-Surplus/Optimise Utilisation achieves better auction surplus (Figure 4.8) even bettering Maximise Surplus. The results (Table 4.5) show that Max-Surplus/Optimise Utilisation (k=0.25) achieves 9% better ask budget met, 11.5% less unsold storage and 12% less unfeasible bids than Maximise Surplus. We can see from the percentage plots that trading off auction surplus for better utilisation achieves much better storage utilisation which in turn achieves a greater auction surplus.

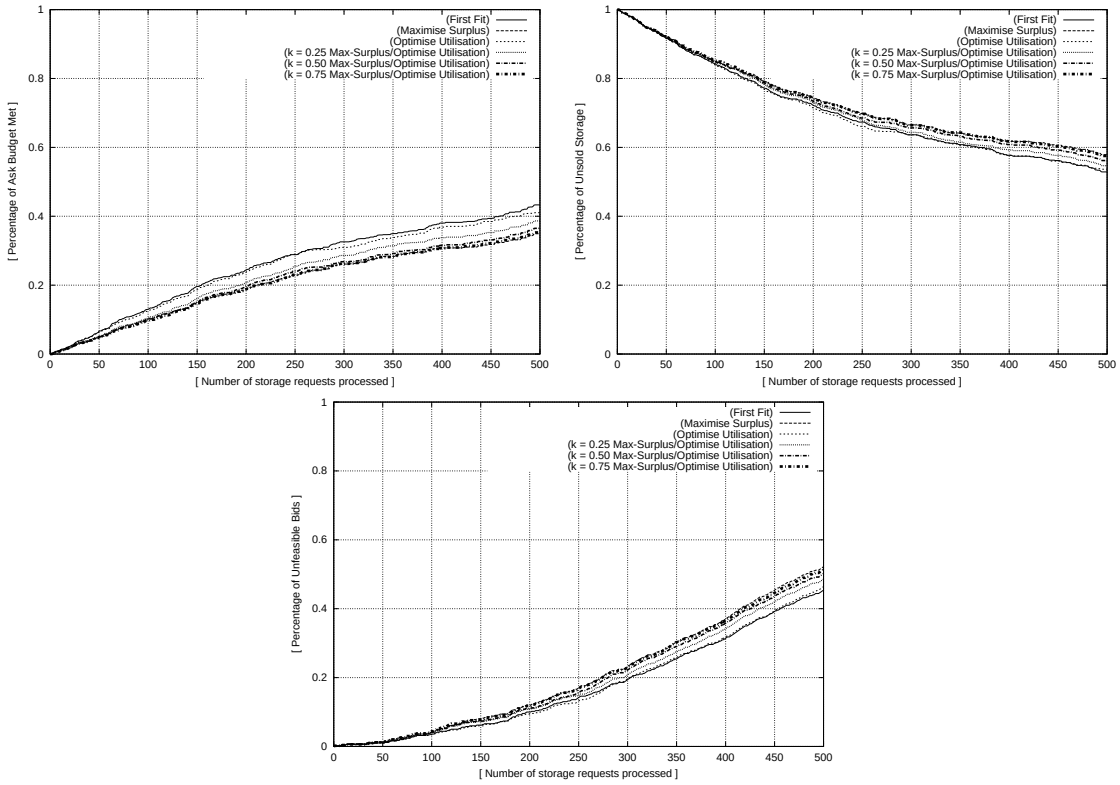

Figure 4.7: Scenario A: percentage of ask budget met, unsold storage and unfeasible bids

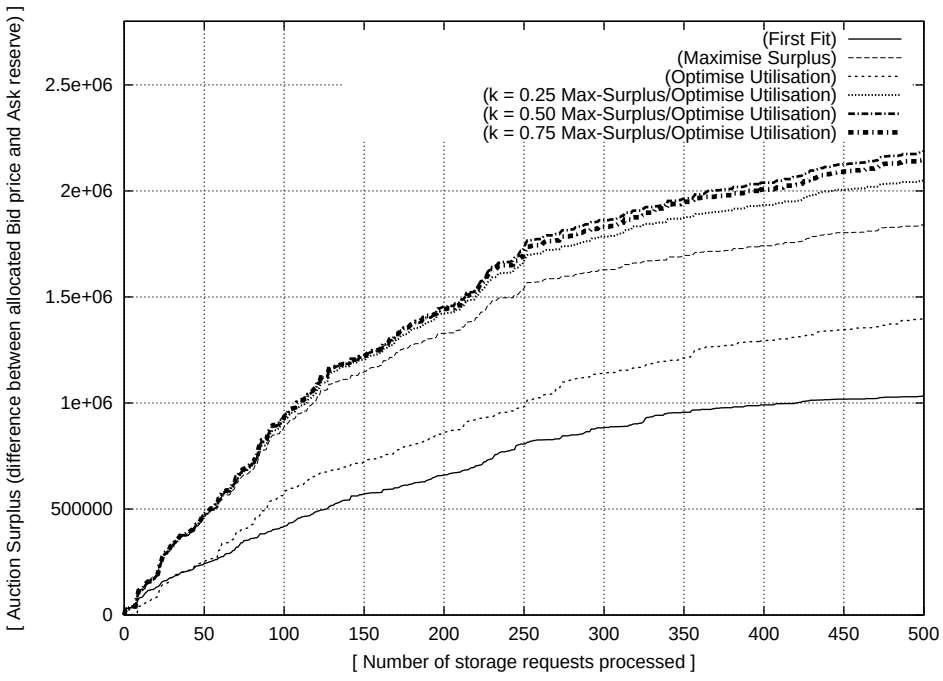

Figure 4.8: Scenario B: auction surplus

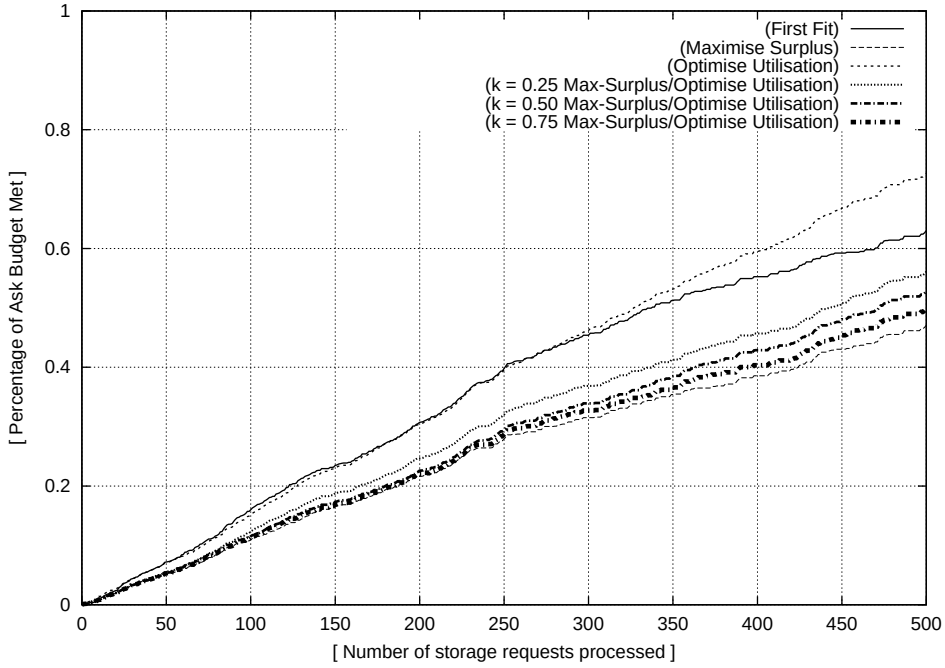

Figure 4.9: Scenario B: percentage of ask budget met

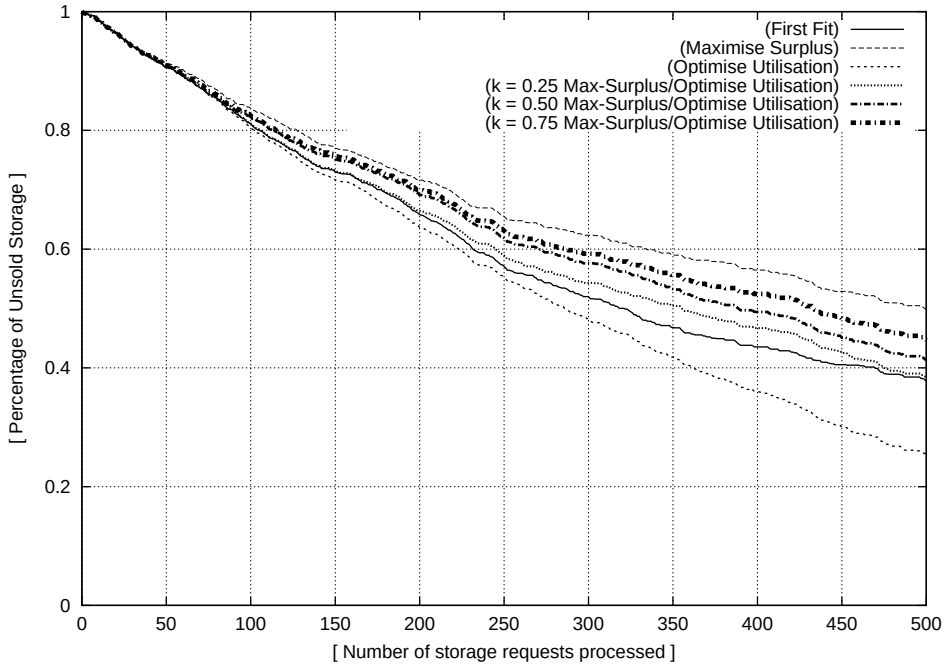

Figure 4.10: Scenario B: percentage of unsold storage

Scenario C

The results in this scenario follow a similar pattern to that of scenario B with Max-Surplus/Optimise Utilisation achieving better results (higher ask budget met, lower rate of unsold storage and unfeasible bids) than Maximise Surplus for all values of k.

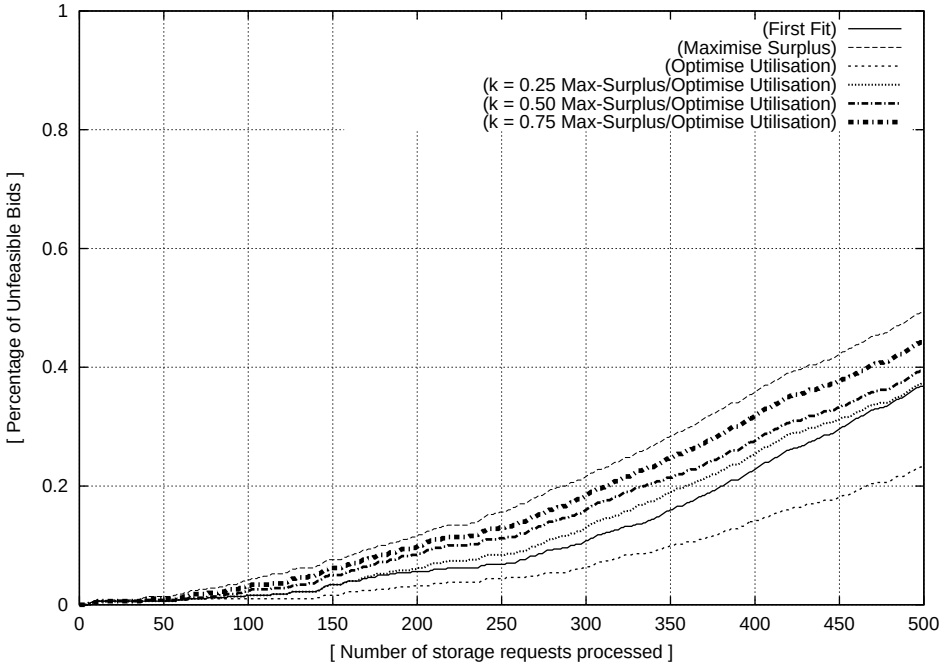

Figure 4.11: Scenario B: percentage of unfeasible bids

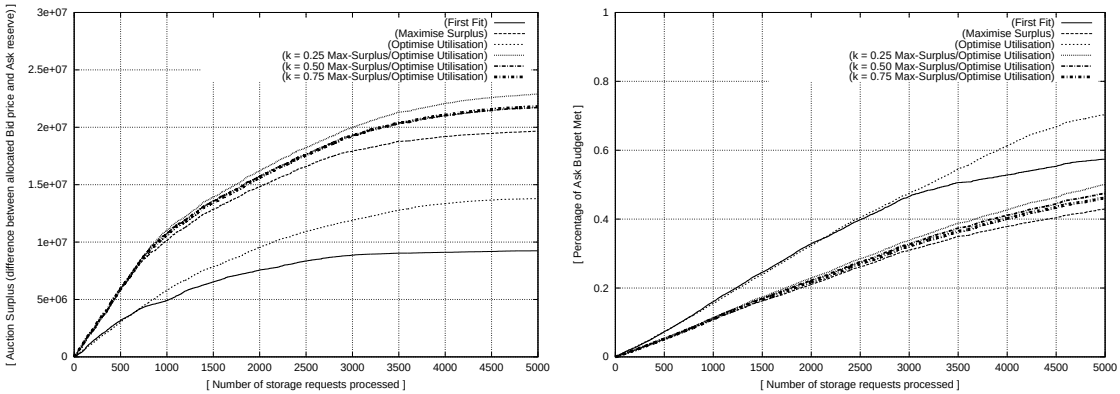

Figure 4.12: Scenario C: auction surplus and percentage of ask budget met

However, when comparing this scenario with scenario B there were subtle differences with Optimise Utilisation achieving 1% less unfeasible bids in Scenario B, but this was contradicted by 1.9% more storage remaining unsold in scenario B than in this scenario, suggesting that a greater rate of capacity was sold across less bids than in scenario B. This contradiction held for Max-Surplus/Optimise Utilisation (for all values of k) with higher unsold storage and lower unfeasible bids in Scenario B than in this scenario. With results between Scenario B and C so similar the Optimise Utilisation seems to have a similar impact, suggesting that it remains effective even when there is a large difference in service attributes ($SSA_{SPA} >> SRB_{SPA}$).

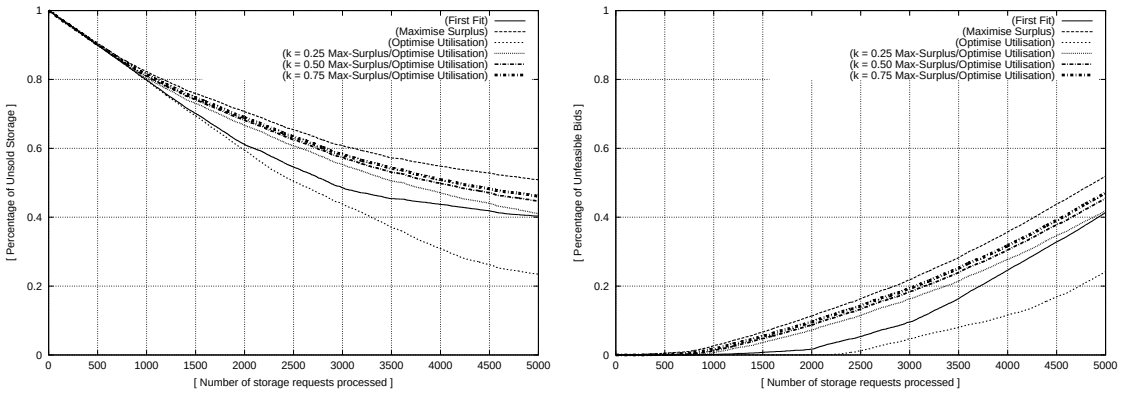

Figure 4.13: Scenario C: percentage of unsold storage and percentage of unfeasible bids

4.7 Summary

When applying a market model to a computer system there are many factors which need to be taken into consideration. These include but are not limited to: architecture, communication overhead, clearing complexity, market efficiency and understanding the nature of the goods being traded. This chapter follows the process of selecting a market model, proposing clearing algorithms and evaluating them through simulations. In the investigation of various auction models, we found the sealed Double Auction market protocol to be effective and efficient, with a low communication overhead allowing multiple transactions to be cleared in an instant. It allows both consumers and providers to submit their exact requirements and its responsibility is to match bids and asks and allocate trades.

The application of sealed Double Auctions in practice is limited to dealing with divisible single attribute goods which do not possess assignment constraints (e.g. shares and currency). Distributed storage, as a tradeable entity in the Storage Exchange, contains many attributes and carries a clearing constraint where demand is indivisible. The process of optimally clearing goods of this nature, in a Double Auction, normally requires solving an NP-hard problem and thus is considered computationally intractable. This chapter proposes algorithms to Optimise Utilisation, Maximise Surplus and by combining these proposes the Max-Surplus/Optimise Utilisation algorithm. The Max-Surplus/Optimise Utilisation algorithm is able to clear trades with indivisible demand constraints in polynomial

time achieving promising empirical results (Scenario 2, k=0.25, 56.1% of ask budget met, 38.3% of storage capacity remain unsold and 37.2% of bids are unfeasible).

Chapter 5

CONCLUSION

This thesis began as a study of distributed storage systems, categorising the wide array of research from the past and present. The study showed that modern day distributed storage systems have evolved into complex systems providing a variety of services from archiving, publishing and anonymity to federating storage services across geographic and institutional boundaries. Furthermore, distributed storage systems are required to function in an unreliable shared environment such as the Internet, posing challenging operational constraints. To meet the demands of operating across a challenging infrastructure such as the Internet, many advances in architecture, security, routing and consistency have been made. The taxonomy distills the plethora of work on DSS, providing a clearer insight into functionality, architecture, operating environment, usage patterns, security, routing, consistency, autonomic management and federation. Following the taxonomy a survey of unique distributed storage systems served to exemplify topics covered earlier in the taxonomy.

In the process of conducting the taxonomy, a study identified the rapid increase in software complexity as the next major obstacle to face future research and development of IT systems [72]. A subsequent feature article [80] outlines a vision for autonomic computing, identifying the need for systems to be self governing, lessening the burden of complexity imposed on administrators and developers. These works laid the foundations for the research and proposal of the Storage Exchange platform. The Storage Exchange applies a market model to automatically allocate storage services based on consumer and provider requirements. The market model sits at the core of the Storage Exchange and the process of selecting an efficient and suitable market model forms much of the research in this thesis. The process of applying a market model to the trading of distributed storage involves:

1. **Understanding the Goods being Traded:** Distributed storage, as a tradeable entity, contains multiple attributes, collectively defined as Storage Policy Attributes (capacity, upload rate, download rate and duration). Whilst further attributes such as replication, consistency even attributes regarding rates of availability could be incorporated into the storage policy attributes, these place further constraints and complicate the process of trading.

2. **Selecting a Market Model:** Before a market model can be successfully applied to a computer system, it is important to consider economic efficiency, communication overhead, clearing complexity and the architectural requirements it may have on the system. Sealed Double Auctions are used widely in practice, known to be economically efficient and due to their ability to clear multiple transactions at an instant, possess remarkably low communication overheads. DAs require a central entity to oversee the trading process and whilst this is a limitation to scalability, it eliminates the need for providers and consumers to search for suitable trades.

   In practice Double Auctions are limited to trading goods with single attributes that are divisible. Although Kalagnanam [79] shows optimally clearing goods where demand is indivisible is possible, it is an NP-hard problem. This poses a dilemma for the Storage Exchange, as not only do storage policies contain multiple attributes, the demand is indivisible as storage requests for a volume may only be serviced by a single provider. This motivated research into a clearing algorithm that is computationally feasible.

3. **Clearing Algorithms:** This thesis presents a polynomial time clearing algorithm which balances auction surplus with optimising utilisation. Simulation results are promising, showing improved levels of utilisation, resulting in best overall auction surplus.

Research into the Storage Exchange platform has identified many challenges facing autonomic management of distributed storage. In the process of employing a market approach to automatically allocate storage services, important realisations

were made to allow distributed storage services to become a tradeable entity. Research presented in this thesis takes a step towards realising an autonomic storage system and is a foundation for future research into employing a market approach to achieve this objective.

5.1 Lessons Learnt About Research

The Storage Exchange has the potential to be a large, all-time consuming system. With so many components, it is easy to be carried away with issues relating to consistency, security, protocol design and multi-threaded design. All exciting topics to investigate and engineer, and all too easy to get side tracked with.

Whilst initial intentions were to build this platform in its entirety, right from providing a mount point to submitting bids and asks to the Storage Marketplace, this soon proved to be an overly ambitious goal. The Storage Broker, Storage Provider and Storage Client and the interactions between these components are functionally complete, the communication between the Storage Marketplace and Storage Broker however are not. Whilst the passion to engineer the system never dulled, due to time constraints, a more rapid simulation approach was used to evaluate the clearing algorithms. During simulation, the Storage Marketplace would load bids and asks from file rather than have having a remote Storage Broker connect and relay them via a posting protocol.

This approach proved to be a quick and effective way to test the feasibility of our clearing algorithms and allowed us to further investigate them. With hindsight, more effort should have been applied to the interactions between the Storage Marketplace and the Storage Broker rather than worrying about the details of the storage service itself.

5.2 Future Directions

The work presented in this thesis represents the beginning of a journey of discovery into autonomic management of storage and whilst many important insights were

made, many questions remain unanswered:

1. Determining a clearing price in a DA where demand is indivisible remains open [79]. It is important when setting a price structure that the market remains incentive compatible, fair and efficient [60].

2. An Investigation of how a combinatorial auction [103] could be applied to the Storage Exchange. Whilst this would substantially increase the clearing complexity [127], consumers and providers would have the flexibility of submitting combinations of bids and asks.

3. A conventional DA market model requires a trusted central entity to collect bids and asks and allocate trades. The presence of a central entity in computer systems poses a scalability and reliability bottleneck, the same applies in our system. Research [36] into executing a DA across a peer-to-peer architecture would provide a more scalable and resilient solution. Another option would be to apply a DA market over a byzantine agreement [21].

4. Allowing volumes to be serviced by multiple providers would simplify the clearing process by eliminating the demand indivisible constraint. Although, managing volume spread across multiple institutions would complicate data management and the manner in which operations are executed.

5. This thesis investigates the clearing process of a single clearing cycle. This could be extended to cover a series of clearing periods across a time period where supply and demand and the clearing interval could be made to fluctuate.

6. Whilst the storage policies incorporate duration, the simulations conducted in our investigation assumed a constant duration. Incorporating duration into the clearing process could introduce too much of an assignment constraint. A possible solution would be to introduce coarse grain time allocation, e.g. short, medium and long term duration. Even an investigation into a futures market, where storage is purchased based on expected usage demand, could be interesting and worthwhile.

7. The Max-Surplus/Optimise Utilisation algorithm provides a parameter (k) allowing allocations to be biased towards achieving auction surplus or better optimisation. Biasing allocations completely towards utilisation (k=0) will yield allocations which are economically inefficient, unfair and impractical. This opens the question of how far allocations can be biased away from max-surplus before the market is deemed too inefficient.

8. Research [38] into allowing systems to be configured with high-level objectives could be incorporated into the storage broker, simplifying administration and taking a step towards realising autonomic storage management.

9. The Storage Provider supports a simple mode of replication, whereby volumes are replicated across multiple hosts. Ultimately a volume's storage capacity is limited to a single host and whilst the data structure (segments) to support volumes being stretched across multiple host is present, the Storage Provider does not support it. More flexible methods to distribute and replicate data could be employed, such as DHT [105].

BIBLIOGRAPHY

[1] Stephen Adler. The slashdot effect – an analysis of three Internet publications. 1999.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

[3] Keno Albrecht, Ruedi Arnold, and Roger Wattenhofer. Clippee: A large-scale client/peer system. October 2003.

[4] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[5] Marcos Assuncao and Rajkumar Buyya. An evaluation of communication demand of auction protocols in grid environments. In *Proceedings of the 3rd International Workshop on Grid Economics and Business (GECON 2006)*. World Scientific Press, May 2006.

[6] Rob Barrett, Yen-Yang Michael Chen, and Paul P. Maglio. System administrators are users, too: designing workspaces for managing internet-scale systems. In *CHI '03: CHI '03 extended abstracts on Human factors in computing systems*, pages 1068–1069, New York, NY, USA, 2003. ACM Press.

[7] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The sdsc storage resource broker. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 1998.

[8] Philip A. Bernstein and Nathan Goodman. The failure and recovery problem for replicated databases. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 114–122, New York, NY, USA, 1983. ACM Press.

[9] D. Bindel and S. Rhea. The design of the oceanstore consistency mechanism, 2000.

[10] Bittorrent. http://www.bittorrent.com/protocol.html.

[11] Johannes Blmer, Malik Kalfane, Richard Karp, Marek Karpinski, Michael Luby, and David Zuckerman. An xor-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, Berkeley, USA, Berkley, 1995.

[12] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 34–43, New York, NY, USA, 2000. ACM Press.

[13] Peter J. Braam. The lustre storage architecture. Cluster File Systems Inc. Architecture, design, and manual for Lustre, November 2002. http://www.lustre.org/docs/lustre.pdf.

[14] Rajkumar Buyya. The virtual laboratory project: Molecular modeling for drug design on grid. In *IEEE Distributed Systems Online*, 2001.

[15] Rajkumar Buyya. *Economic-based distributed resource management and scheduling for Grid computing*. PhD thesis, Monash University, Melbourne, Australia, 2002.

[16] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[17] Alessandra Cassar and Daniel Friedman. An electronic calendar auction: White paper. Technical report, University of California, April 2000.

[18] Castro and Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[19] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks, 2002.

[20] Miguel Castro, Atul Adya, Barbara Liskov, and Andrew C. Myers. HAC: Hybrid adaptive caching for distributed storage systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 102–115, Saint Malo, France, October 1997.

[21] Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-Fault-Tolerant system. In *In Proc. of Sigmetrics*, pages 273–288, June 2000.

[22] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. In *Journal of Network and Computer Applications*, volume 23, July 2000.

[23] Brent N. Chun, Jeannie Albrecht, David C. Parkes, and Amin Vahdat. Computational resource exchanges for distributed resource allocation. http://citeseer.ist.psu.edu/706369.html, 2005.

[24] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[25] Brian F. Cooper, Arturo Crespo, and Hector Garcia-Molina. The stanford archival repository project: Preserving our digital past, 2002.

[26] Brian F. Cooper and Hector Garcia-Molina. Peer-to-peer data trading to preserve information. *ACM Transactions on Information Systems*, 20(2):133–170, 2002.

[27] Brian F. Cooper and Hector Garcia-Molina. Peer-to-peer data preservation through storage auctions. *IEEE Transactions Parallel Distributed Systems*, 16(3):246–257, 2005.

[28] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, San Diego, CA, 1995. IEEE Computer Society Press.

[29] Simon Cuce and Arkady B. Zaslavsky. Adaptable consistency control mechanism for a mobility enabled file system. In *MDM '02: Proceedings of the Third International Conference on Mobile Data Management*, pages 27–34, Washington, DC, USA, 2002. IEEE Computer Society.

[30] K. Bill D. Dimitri, G. Antonio. Analysis of peer-to-peer network security using gnutella. 2002.

[31] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common api for structured peer-to-peer overlays, 2003.

[32] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[33] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 2–15, May 2002.

[34] C. J. Date. *Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[35] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, Santa Cruz, California, 8-9 1994.

[36] Zoran Despotovic, Jean-Claude Usunier, and Karl Aberer. Towards peer-to-peer double auctioning. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, page 90289.1, Washington, DC, USA, 2004. IEEE Computer Society.

[37] Murthy Devarakonda, Alla Segal, and David Chess. A toolkit-based approach to policy-managed storage. In *POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, page 89, Washington, DC, USA, 2003. IEEE Computer Society.

[38] Murthy V. Devarakonda, David M. Chess, Ian Whalley, Alla Segal, Pawan Goyal, Aamer Sachedina, Keri Romanufa, Ed Lassettre, William Tetzlaff, and Bill Arnold. Policy-based autonomic storage allocation. In Marcus Brunner and Alexander Keller, editors, *DSOM*, volume 2867 of *Lecture Notes in Computer Science*, pages 143–154. Springer, 2003.

[39] Sarana Nutanong Ding Choon-Hoong and Rajkumar Buyya. Peer-to-peer networks for content sharing. In Ramesh Subramanian and Brian Goodman, editors, *Peer-to-Peer Computing: Evolution of a Disruptive Technology*, pages 28–65. Idea Group Publishing, Hershey, PA, USA, 2005.

[40] Roger Dingledine. The free haven project: Design and deployment of an anonymous secure data haven. Master's thesis, MIT, June 2000.

[41] Roger Dingledine, Michael J. Freedman, and David Molnar. The free haven project: Distributed anonymous storage service. In *Workshop on Design Issues in Anonymity and Unobservability*, number 2009 in LNCS, pages 67–95, 2000.

[42] Roger Dingledine, Nick Mathewson, and Paul Syverson. Reputation in p2p anonymity systems, June 2003.

[43] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the Seventh USENIX Security Symposium*, August 2004.

[44] J. Douceur. The sybil attack, 2002.

[45] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 59–70. ACM Press, 1999.

[46] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, pages 75–80, Schloss Elmau, Germany, May 2001.

[47] Patrick Eaton and Steve Weis. Examining the security of a file system interface to oceanstore.

[48] EncFS. http://encfs.sourceforge.net/, 2000.

[49] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[50] Dror G. Feitelson. On the scalability of centralized control. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18*, page 298.1, Washington, DC, USA, 2005. IEEE Computer Society.

[51] M. Feldman, K. Lai, J. Chuang, and I. Stoica. Quantifying disincentives in peer-to-peer networks. In *1st Workshop on Economics of Peer-to-Peer Systems*, 2003.

[52] Donald F. Ferguson, Christos Nikolaou, Jakka Sairamesh, and Yechiam Yemini. Economic models for allocating resources in computer systems. pages 156–183, 1996.

[53] Ian T. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 1–4, London, UK, 2001. Springer-Verlag.

[54] Michael J. Freedman and Robert Morris. Tarzan: a peer-to-peer anonymizing network layer. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 193–206, New York, NY, USA, 2002. ACM Press.

[55] Daniel Friedman and John Rust. *The Double Auction Market: Institutions, Theories and Evidence*. Addison-Wesley Publishing, 1993.

[56] FUSE. http://sourceforge.net/projects/fuse/, 2000.

[57] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43. ACM Press, 2003.

[58] Deepinder S. Gill, Songnian Zhou, and Harjinder S. Sandhu. A case study of file system workload in a large-scale distributed environment. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 276–277. ACM Press, 1994.

[59] Steven Gjerstad and John Dickhaut. Price formation in double auctions. In *E-Commerce Agents, Marketplace Solutions, Security Issues, and Supply and Demand*, pages 106–134, London, UK, 2001. Springer-Verlag.

[60] Steven Gjerstad and John Dickhaut. Price formation in double auctions. In Jiming Liu and Yiming Ye, editors, *E-Commerce Agents*, volume 2033 of *Lecture Notes in Computer Science*, pages 106–134. Springer, 2001.

[61] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. pages 181–208, 1994.

[62] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[63] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. volume 15, pages 287–317, New York, NY, USA, 1983. ACM Press.

[64] Garrett Hardin. Tragedy of the commons. *Science*, 162(3859):1243–1248, 1968.

[65] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks, 2002.

[66] Anthony Harrington and Christian Jensen. Cryptographic access control in a distributed file system. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 158–165, New York, NY, USA, 2003. ACM Press.

[67] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 309–329. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[68] Ragib Hasan, Zahid Anwar, William Yurcik, Larry Brumbaugh, and Roy Campbell. A survey of peer-to-peer storage techniques for distributed file systems. In *IEEE International Conference on Information Technology (ITCC)*. IEEE, April 2005.

[69] S. Hazel and B. Wiley. Achord: A variant of the chord lookup service for use in censorship resistant peer-to-peer publishing systems, 2002.

[70] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software Practice and Experience*, 28(9):901–928, 1998.

[71] K. Holtman. Cms data grid system overview and requirements. 2001.

[72] P. Horn. Autonomic computing: Ibm's perspective on the state of information technology, October 2001.

[73] Wolfgang Hoschek, Francisco Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (GRID '00)*, Bangalore, India, December 2000. Springer-Verlag, London, UK.

[74] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions Computing Systems*, 6(1):51–81, 1988.

[75] D. Hughes, G. Coulson, and J.Walkerdine. Freeriding on gnutella revisited: the bell tolls? In *IEEE Distributed Systems Online*, 2005.

[76] IEEE/ANSI Std. 1003.1. *Portable operating system interface (POSIX)-part 1: System application program interface (API) [C language], 1996 editon.*

[77] Jr. James V. Huber, Andrew A. Chien, Christopher L. Elford, David S. Blumenthal, and Daniel A. Reed. Ppfs: a high performance portable parallel file system. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 385–394, New York, NY, USA, 1995. ACM Press.

[78] M B Jones. Web-based data management. In S.G Stafford W.K Michener, J.H Porter, editor, *Data and Information Management in the Ecological Sciences: A resource Guide*, Albuquerque, New Mexico, 1998. University of New Mexico.

[79] Jayant R. Kalagnanam, Andrew J. Davenport, and Ho S. Lee. Computational aspects of clearing continuous call double auctions with assignment constraints and indivisible demand. *Electronic Commerce Research*, 1(3):221–238, 2001.

[80] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[81] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.

[82] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[83] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[84] Zhenmin Li, Zhifeng Chen, and Yuanyuan Zhou. Mining block correlations to improve storage performance. *Transactions on Storage*, 1(2):213–245, 2005.

[85] Zhenmin Li, Sudarshan M. Srinivasan, Zhifeng Chen, Yuanyuan Zhou, Peter Tzvetkov, Xifeng Yan, and Jiawei Han. Using data mining for discovering patterns in autonomic storage systems.

[86] Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, pages 72–93, 2005.

[87] Nancy A. Lynch, Dahlia Malkhi, and David Ratajczak. Atomic data access in distributed hash tables. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 295–305, London, UK, 2002. Springer-Verlag.

[88] Sergio Marti and Hector Garcia-Molina. Identity crisis: Anonymity vs. reputation in p2p systems. In *Peer-to-Peer Computing*, pages 134–141. IEEE Computer Society, 2003.

[89] Shigeo Matsubara. Accelerating information revelation in ascending-bid auctions: avoiding last minute bidding. In *EC '01: Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 29–37, New York, NY, USA, 2001. ACM Press.

[90] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[91] R Preston McAfee and John McMillan. Auctions and bidding. *Journal of Economic Literature*, 25(2):699–738, June 1987.

[92] Lee W. McKnight and Jahangir Boroumand. Pricing internet services: Approaches and challenges. *Computer*, 33(2):128–129, 2000.

[93] Kumar Mehta and Byungtae Lee. An empirical evidence of winner's curse in electronic auctions. In *ICIS '99: Proceeding of the 20th international conference on Information Systems*, pages 465–471, Atlanta, GA, USA, 1999. Association for Information Systems.

[94] Ayse Morali, Leonardo Varela, and Carlos Varela. An electronic marketplace: Agent-based coordination models for online auctions. In *XXXI Conferencia Latinoamericana de Informática*, Cali, Colombia, October 2005.

[95] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, 1986.

[96] Steven A. Moyer and V. S. Sunderam. PIOUS: A scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[97] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*. USENIX, December 2002.

[98] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. April 1995. Supersedes FIPS PUB 180 1993 May 11.

[99] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, 1996. ACM Press.

[100] Brian D. Noble and M. Satyanarayanan. *An empirical study of a highly available file system*. New York, NY, USA, 1994.

[101] Elth Ogston and Stamatis Vassiliadis. A peer-to-peer agent auction. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 151–159, New York, NY, USA, 2002. ACM Press.

[102] Andy Oram. *Peer-to-Peer : Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Sebastopol, CA, 2001.

[103] Aleksandar Pekec and Michael H. Rothkopf. Combinatorial auction design. *Manage. Sci.*, 49(11):1485–1503, 2003.

[104] Martin Placek and Rajkumar Buyya. Storage exchange: A global trading platform for storage services. In *12th International European Parallel Computing Conference (EuroPar)*, LNCS, Dresden, Germany, August 2006. Springer-Verlag, Berlin, Germany.

[105] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[106] Arcot Rajasekar, Michael Wan, and Reagan Moore. Mysrb & srb: Components of a data grid, 2002.

[107] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[108] Daniel A Reed, Celso L Mendes, Chang da Lu, Ian Foster, and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure - Application Tuning and Adaptation*. Morgan Kaufman, San Francisco, CA, second edition, 2003. pp.513-532.

[109] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of the Conference on File and Storage Technologies*. USENIX, 2003.

[110] Richard G. Lipsey and K.Alec Chrystal. *Principles of Economics 9th Edition*. Oxford University Press, 1999.

[111] Daniel R. Ries and Michael Stonebraker. Effects of locking granularity in a database management system. *ACM Transactions on Database Systems*, 2(3):233–246, 1977.

[112] Daniel R. Ries and Michael R. Stonebraker. Locking granularity revisited. *ACM Transactions on Database Systems*, 4(2):210–227, 1979.

[113] R. L. Rivest. The MD5 Message Digest Algorithm. RFC 1321, April 1992.

[114] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[115] Alvin E. Roth and Axel Ockenfels. Last-minute bidding and the rules for ending second-price auctions: Evidence from ebay and amazon auctions on the internet. *American Economic Review*, 92(4):1093–1103, 2002.

[116] Michael H Rothkopf and Ronald M Harstad. Two models of bid-taker cheating in vickrey auctions. *Journal of Business*, 68(2):257–67, April 1995.

[117] Michael H Rothkopf, Thomas J Teisberg, and Edward P Kahn. Why are vickrey auctions rare? *Journal of Political Economy*, 98(1):94–109, February 1990.

[118] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, volume 2218, pages 329–350, November 2001.

[119] B. Rudis and P. Kostenbader. The enemy within: Firewalls and backdoors, Jun 2003.

[120] Aldo Rustichini, Mark A Satterthwaite, and Steven R Williams. Convergence to efficiency in a simple market with incomplete information. *Econometrica*, 62(5):1041–63, 1994.

[121] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[122] Tuomas W. Sandholm. Distributed rational decision making. In Gerhard Weiß, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 201–258. MIT Press, Cambridge, MA, USA, 1999.

[123] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[124] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–18, 20–21, 1990.

[125] Mahadev Satyanarayanan. The influence of scale on distributed file system design. *IEEE Transactions on Software Engineering*, 18(1):1–8, 1992.

[126] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, January 2002.

[127] B. Schnizler, D. Neumann, and C.Weinhardt. Resource allocation in computational grids - a market engineering approach. In *In: Proceedings of the WeB*, Washington, US, 2004.

[128] Rüdiger Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Peer-to-Peer Computing*, pages 101–102. IEEE Computer Society, 2001.

[129] Wayne Schroeder. The sdsc encryption/authentication (sea) system. *Concurrency - Practice and Experience*, 11(15):913–931, 1999.

[130] S. T. Shafer. Corporate espionage the enemy within. Red Herring, January 2002.

[131] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables, 2002.

[132] Vernon L. Smith. An experimental study of competitive market behavior. *The Journal of Political Economy*, 70(2):111–137, April 1962.

[133] Mirjana Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems*, 14(2):200–222, 1996.

[134] Steffen Staab, Francis Heylighen, Carlos Gershenson, Gary William Flake, David M. Pennock, Daniel C. Fain, David De Roure, Karl Aberer, Wei-Min Shen, Olivier Dousse, and Patrick Thiran. Neurons, viscose fluids, freshwater polyp hydra-and self-organizing information systems. *IEEE Intelligent Systems*, 18(4):72–86, 2003.

[135] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[136] Rethinking Enterprise Storage Strategies. Cio custom publishing, October 2002.

[137] P F Syverson, D M Goldschlag, and M G Reed. Anonymous connections and onion routing. In *IEEE Symposium on Security and Privacy*, pages 44–54, Oakland, California, 4–7 1997.

[138] Richard Thaler. *Winner's Curse : Paradoxes and anomalies of economic life (Russell Sage Foundation Study)*. Free Press, December 1991.

[139] Chandramohan A. Thekkath and Edward K. Lee. Petal: Distributed virtual disks. In *Proc. 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, October 1996.

[140] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

[141] Peter Triantafillou and Theoni Pitoura. Towards a unifying framework for complex query processing over structured peer-to-peer data networks. In Karl Aberer, Vana Kalogeraki, and Manolis Koubarakis, editors, *DBISP2P*, volume 2944 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2003.

[142] Kurt Tutschku. A measurement-based traffic profile of the edonkey filesharing service. In Chadi Barakat and Ian Pratt, editors, *PAM*, volume 3015 of *Lecture Notes in Computer Science*, pages 12–21. Springer, 2004.

[143] Sudharshan S. Vazhkudai, Xiaosong Ma, Vincent W. Freeh, Jonathan W. Stricklandand Nandan Tammineedi, , and Stephen L. Scott. Freeloader: Scavenging desktop storage resources for scientific data. In *IEEE/ACM Supercomputing 2005 (SC—05)*, Seattle, WA, November 2005. IEEE Computer Society.

[144] Srikumar Venugopal, Rajkumar Buyya, and Kotagiri Ramamohanarao. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Computing Survey*, 28, Mar 2006.

[145] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 16(1):8–37, 1961.

[146] M. Waldman, A.D. Rubin, and L.F. Cranor. Publiues: a robust tamper-evident censorship-resistant web publishing system. In *Proceedings of the Nineth USENIX Security Symposium*, Denver, CO, USA, 2000. USENIX Association.

[147] Ruqu Wang. Auctions versus posted-price selling. *American Economic Review*, 83(4):838–51, September 1993.

[148] H. Weatherspoon, C. Wells, P. Eaton, B. Zhao, and J. Kubiatowicz. Silverback: A global-scale archival system, 2001.

[149] Jan Weglarz, Jarek Nabrzyski, and Jennifer Schopf, editors. *Grid resource management: state of the art and future trends*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[150] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM Press.

[151] Bryce Wilcox-O'Hearn. Experiences deploying a large-scale emergent network. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 104–110. Springer-Verlag, 2002.

[152] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan. G-commerce: Market formulations controlling resource allocation on the computational grid. In *International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, April 2001. IEEEE.

[153] Mao Yang, Zheng Zhang, Xiaoming Li, and Yafei Dai. An empirical study of free-riding behavior in the maze p2p file-sharing system. In *4th International Workshop on Peer-To-Peer Systems*. Ithaca, New York, USA, February 2005.

[154] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D Joseph, and John D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003. Special Issue on Service Overlay Networks, to appear.

Appendix A

STORAGE BROKER DATA DICTIONARY

Table A.1: user table description

Table Name: User	
Field Name	*Description*
userINDEX	Primary key.
userLoginName	Users login name.
userPassword	Password used to login.
userType	External or Internal: External users a limited to only viewing Virtual Volume information. Internal users are allowed to add new entries in the available storage table and create Virtual Volumes.

Table A.2: available storage table description

Table Name: AvailableStorage	
Field Name	*Description*
availableStoreINDEX	Primary Key.
userINDEX	Owner of available storage.
entityID	Unique identifier of available store. If isContract field is true than this field represents the ContractID otherwise its the StorageEntityID.
contactHostIP	Is the host's IP address responsible for servicing this available store. If isContract field is true then this field represents the Storage Broker IP, otherwise this field storage the Storage Provider IP.
contactHostListenPortNumber	Is the portnumber of the host responsible for servicing this available store.
Capacity_MB	Device storage capacity in Megabytes.
Used_MB	Raw used storage on device in Megabytes.
Free_MB	Available storage on device in Megabytes.
UploadRate_kB	Maximum allowable upload rate in Kilobytes. if 0, then no limit.
DownloadRate_kB	Maximum allowable download rate in Kilobytes. if 0, then no limit.
IsContract	If true, this record is a storage contract and will have a reference to the contract table containing contract specific attributes, otherwise this record represents a storage provider.
Status	Only applicable if isContract is false, that is we are dealing with a local Storage Provider. The status of a Storage Provider is considered "Available" if it is currently connected to the Storage Broker, otherwise it is flagged as " Unavailable".

Table A.3: contract table description

Table Name: Contract	
Field Name	*Description*
availableStoreINDEX	Foreign key, used to reference the available storage table.
AllocatedBudget	Budget allocated to purchase contract.
ContractCost	The actual negotiated cost of acquiring this contract, must be <= to allocated budget.
Duration	Contract lifetime in seconds.

Table A.4: virtual volume table description

Table Name: Virtual Volume	
Field Name	*Description*
virtualVolumeINDEX	Primary Key.
userINDEX	Index to user which owns this Vitual Volume.
VolumeID	Name of the Volume.
Capacity_MB	Virtual Volume storage capacity in Megabytes..
UploadRate_kB	Maximum allowable upload rate in Kilobytes. if 0, then no limit.
DownloadRate_kB	Maximum allowable download rate in Kilobytes. if 0, then no limit.
Duration	Virtual Volume duration in seconds.
replicationLevel	Replication level for this volume. Each segment allocated to this volume will inherit this replication level.
isForSale	If false, Virtual Volume is not be sold, most probably as it is to be used within the institution, otherwise Virtual Volume is to be put up for trade.
askBudget	If isForSale is true, than this field represents the asking price for the service.
sellStatus	If isForSale is true, than this field determines if this volume has been "sold" or remains "unsold".
numUsers	Used to limit the number of users accessing Virtual Volume. This field has been added in for future functionality, where if numUsers is 1 then a weak approach to consistency could be applied, otherwise if numUsers > 1 than a stronger approach to consistency will need to be applied.

Table A.5: segment table description

Table Name: Segment	
Field Name	*Description*
segmentINDEX	Primary key.
virtualVolumeINDEX	Foreign key.
Mode	Can be one of replication, stripe (unimplemented), DHT (unimplemented).
Capacity_MB	Segment storage capacity in Megabytes.
Used_MB	Segment used storage capacity in Megabytes, to be reported by Storage Provider.

Table A.6: segment available store table description

Table Name: Segment Available Store	
Field Name	*Description*
segmentAvailableStoreINDEX	Primary key.
segmentINDEX	Foreign key.
availableStoreINDEX	Foreign key.

Appendix B

STORAGE EVENT PROTOCOL

The Storage Event Protocol is used for all the communication amongst the components which make up the Storage Exchange platform. In this section we discuss the details the protocol behaviour and the structure of each storage event message.

B.1 Storage Event Message

Each Storage Event message comprises of a header and a payload. The header contains a fixed number of attributes which are globally required in every Storage Event Message. The payload on the other hand is a variable length field which itself may contain many fields depending on the message type specified by the header.

HEADER				**BODY**
`MessageType`	`UniqueID`	`Length`	`ConnID`	`Payload`
4 bytes	*4 bytes*	*4 bytes*	*4 bytes*	*specified by the Length field*

Table B.1: storage event message structure

B.1.1 Header

The header comprises of the following four 32 bit fields:

1. *Message Type:* This field is used to determine the message type. We describe each of the possible message types in Section B.2.

2. *Unique ID:* Most of the communication consists of two messages being exchanged. A request is sent and a reply is expected. The Unique ID field is used to uniquely identify a pair of request and reply messages. This is particularly useful when dealing with asynchronous communication which is the case between the Storage Client and Storage Provider components.

3. *Length:* Length of the payload.

4. *ConnID:* A unique identifier representing the connection id between two components, the id is assigned by the party that accepted the connection. If the assigned connection ID is less than 0 than there is a problem with how the two components have handshaked (Section B.2.1).

B.1.2 Payload

Payload can be arbitrary length and may contain many fields of arbitrary types. The message type field in the header can be used to determine what fields are to be expected and the length field in the header determines the length of the payload.

B.2 Storage Event Types

There three categories of Storage Events (i) Handshakes - used when any two components initiate communication, (ii) Trading Protocol - used by Storage Broker and Storage Marketplace to exchange trade information. (iii) Storage Protocol - used between the Storage Client and Storage Provider.

B.2.1 Handshakes

A pair of handshake Storage Event messages are exchanged anytime two components establish a connection. When a connection is established, the party that initiated the connection (A party) is responsible for sending a sign-on storage event, which the receiving party (B party) replies to with a reply storage event. There are five different types of handshakes, which include:

Storage Client and Storage Broker

This handshake is used when a Storage Client initiates a connection to a Storage Broker, as part of the process of mounting a Virtual Volume (Section 3.2.6 : Step 1). This handshake (Table B.2) is used by the Storage Client to authenticate itself with the Storage Broker. Upon successful authentication the Storage Client is able to send request to mount volume for the specified Virtual Volume for servicing.

Storage Client sends storage event with the following structure:				
HEADER				**BODY**
`MessageType`	`UniqueID`	`Length`	`ConnID`	`Payload`
52	*specified*	*specified*	*Unused*	*User ID*

Storage Broker replies with a storage event following this structure:				
HEADER				**BODY**
`MessageType`	`UniqueID`	`Length`	`ConnID`	`Payload`
52	*specified*	*specified*	*specified*	*Error Message*
If connID > 0, than storage client has been successfully authenticated. If connID $= 0$, than error and an error message will be specified.				

Table B.2: storage client and storage broker handshake

Primary Provider and Secondary Provider

This handshake is used when a Primary Provider initiates a connection to a Secondary provider, as part of the process of mounting a Virtual Volume (Section 3.2.6 : Step 3). This handshake (Table B.3) is used to notify the Secondary Provider of the Virtual Volume and Segment it will be servicing.

Primary Provider and Storage Client

This handshake is used when a Primary Provider initiates a connection to the Storage Client, as part of the process of mounting a Virtual Volume (Section 3.2.6 : Step 4). This handshake (Table B.4)is used to notify the Storage Client the Primary Provider is ready to service the Virtual Volume.

Storage Provider and Storage Broker

This handshake is used by all Storage Provider to register and connect with the Storage Broker. A Storage Provider needs to be registered and connected with the Storage Broker to receive a request to mount a Virtual Volume (Section 3.2.6 : Step 2). This handshake is used to inform

Primary Storage Provider sends storage event with the following structure:				
HEADER				**BODY**
MessageType	UniqueID	Length	ConnID	Payload
54	*specified*	*specified*	*Unused*	*StorageEntityID,VolumeID,SegmentID*
Secondary Storage Provider replies with a storage event following this structure:				
HEADER				**BODY**
MessageType	UniqueID	Length	ConnID	Payload
54	*specified*	*specified*	*specified*	*Error Message*
If connID > 0, than handshake was successful. If connID = 0, than error and an error message will be specified.				

Table B.3: primary storage provider and secondary storage provider handshake

Primary Storage Provider sends storage event with the following structure:				
HEADER				**BODY**
MessageType	UniqueID	Length	ConnID	Payload
53	*specified*	*specified*	*Unused*	*StorageEntityID*
Storage Client replies with a storage event following this structure:				
HEADER				**BODY**
MessageType	UniqueID	Length	ConnID	Payload
53	*specified*	*specified*	*specified*	*Error Message*
If connID > 0, than handshake was successful. If connID = 0, than error and an error message will be specified.				

Table B.4: primary storage provider and storage client handshake

the Storage Broker of the Primary Provider's storage potential and provider listen port number, allowing primary providers to connect to it (Table B.5).

Storage Provider sends storage event with the following structure:				
HEADER				**BODY**
MessageType	UniqueID	Length	ConnID	Payload
51	*specified*	*specified*	*unused*	*UserID,StorageEntityID,* *ProvListenPortNum,* *diskFreeSpace,diskCapacity*
If StorageEntityID=0 then provider is signing on for the first time and reply payload will specify its StorageEntityID.				
Storage Broker replies with a storage event following this structure:				
HEADER				**BODY**
MessageType	UniqueID	Length	ConnID	Payload
51	*specified*	*specified*	*specified*	*StorageEntityID*
If connID > 0, than handshake was successful, and if its the first sign on the payload will contain unique StorageEntityID If connID = 0, than error and an error message will be specified in the payload.				

Table B.5: storage provider and storage broker handshake

Storage Broker and Storage Marketplace

Protocol used between the Storage Broker and Storage Marketplace is incomplete[1].

[1]Refer to Lessons Learnt About Research for details.

B.2.2 Trading Protocol

Trading Protocol used between the Storage Broker and Storage Marketplace is incomplete.

B.2.3 Storage Protocol

The Storage Protocol consists of the Storage Client issuing storage requests and for each request the Storage Provider transmits a reply. The protocol is based on the FUSE API [56] which follows system file Input/Output calls found in modern operating systems. In Table B.6 we iterate through each of the message types and outline the attributes contained in both the requesting storage event and the reply storage event. For example: the GETATTR message type is issued by the Storage Client by transmitting a storage request with a payload consisting of the path. Upon receiving the GETATTR storage request the storage provider issues the system *stat* command for the specified path and transmits a storage reply with a payload containing the return code and stat struct. Details of the exact fields within the stat struct and all other structs in Table B.6 are supplied in man pages.

Message Type	Request /Reply	Payload Attributes	Equivalent system call
GETATTR	request	**char** *path	stat
	reply	**int** returnCode, **struct stat** *buf	
READLINK	request	**int** sizeOfBuf, **char** *path	readlink
	reply	**int** returnCode, **char** *buf	
GETDIR	request	**char** *path	readdir
	reply	**int** returnCode, **struct dirent** *buf	
MKNOD	request	**mode_t** mode, **dev_t** dev, **char *path**	mknod
	reply	**int** returnCode	
MKDIR	request	**mode_t** mode, **char** *path	mkdir
	reply	**int** returnCode	
CHMOD	request	**mode_t** mode, **char** *path	chmod
	reply	**int** returnCode	
UNLINK	request	**char** *path	unlink
	reply	**int** returnCode	
RMDIR	request	**char** *path	rmdir
	reply	**int** returnCode	
CHOWN	request	**uid_t** owner, **gid_t** group, **char** *path	chown
	reply	**int** returnCode	
SYMLINK	request	**char** *from, **char** *to	symlink
	reply	**int** returnCode	
RENAME	request	**char** *from, **char** *to	rename
	reply	**int** returnCode	
LINK	request	**char** *from, **char** *to	link
	reply	**int** returnCode	
TRUNCATE	request	**off_t** length, **char** *path	truncate
	reply	**int** returnCode	
UTIME	request	**struct utimbuf** *buf, **char** *path	utime
	reply	**int** returnCode	
STATFS	request	**struct statfs** *buf, **char** *path	statfs
	reply	**int** returnCode	
OPEN	request	**int** flags, **char** *path	open
	reply	**int** returnCode	
READ	request	**size_t** count, **off_t** offset, **char** *path	pread
	reply	**int** returnCode	
WRITE	request	**size_t** count, **off_t** offset, **char** *buf, **char** *path	pwrite
	reply	**int** returnCode	

Table B.6: storage protocol operations