A Taxonomy of Distributed Storage Systems

MARTIN PLACEK and RAJKUMAR BUYYA
The University of Melbourne

Revision : 1.148

This paper presents a taxonomy of key topics effecting research and development of distributed storage systems. The taxonomy finds 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 challenges faced by distributed storage systems and the research to overcome them. The paper continues by providing a survey of distributed storage systems which exemplify topics covered in the taxonomy.

The selection of surveyed systems covers a variety of storage systems, exposing the reader to an array of different problems and solutions employed to overcome these challenges. For each surveyed system we address the underlying operational behaviour, leading into the architecture and algorithms employed in the design and development of the system. Our survey covers systems from the past and present concluding with a discussion on the evolution of distributed storage systems and possible future work.

Categories and Subject Descriptors: E.0 [**Data**]: General—*Distributed Storage systems*

General Terms: Distributed Storage

Additional Key Words and Phrases: distributed storage, distributed architecture, network storage, taxonomy, autonomic, file system

1. INTRODUCTION

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 networking and storage has created a platform with numerous possibilities allowing Distributed Storage Systems (DSS) to adopt roles vast and varied which fall well beyond data storage. This paper discusses DSSs from their inception as Distributed File Systems (DFS) to DSSs capable of spanning a global network of users providing a rich set of services; from publishing, file sharing and high performance to global federation and utility storage.

Networking infrastructure and distributed computing share a close relationship. Advances in networking are typically followed by new distributed storage systems, which better utilise the networks capability. To illustrate, when networks evolved from mostly 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. These next generation Internet based systems are faced with many challenges, including longer delays, unreliability, unpredictability and potentially malicious behaviour, all associated with operating in a public shared environment. To cope with these challenges 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 [Horn

2001]. Unfortunately the continuing increase in complexity is unsustainable and ultimately limited by human cognitive capacity [Staab et al. 2003]. To address this problem the Autonomic Computing [Kephart and Chess 2003] vision has emerged aiming to overcome the ”*complexity crisis*”.

Global connectivity enables resources to be shared amongst individuals and institutions, the processes which make this possible have sparked research into the field of Grid computing [Reed et al. 2003]. Grid computing focuses on solving challenges associated with coordinating and sharing heterogeneous resources across multiple geographic and administrative domains [Foster 2001]. One of these challenges is data management, which has given rise to the Data Grid [Chervenak et al. 2000]. Some of the challenges of managing globally distributed data include providing a standard uniform interface across a heterogeneous set of systems [Rajasekar et al. 2002], coordinating and processing of data [Venugopal et al. 2006] and managing necessary meta-data [Hoschek et al. 2000].

Distributed systems designed to operate on the Internet need to cope with a potential user base numbering in the millions. To accommodate a large user base, distributed systems are employing more scalable decentralised Peer-to-Peer architectures over conventional centralised architectures. Distributed systems which operate across a public network need to accommodate for a variety of potentially malicious behaviour [Douceur 2002; Dingledine 2000] from free-riding [Hughes et al. 2005] to Denial of Service (DoS) attacks [Wilcox-O’Hearn 2002]. Whilst some level of trust can be assumed when operating in a private LAN, this assumption does not hold when connected to a public network and thus algorithms to establish trust and secure data are required.

Distributed storage systems have evolved from providing a means to store data remotely, to offering innovative services like publishing, federation, anonymity and archival. To make this possible networks have evolved to span the globe. With network infrastructure expecting to undergo another quantum leap, outpacing the bandwidth capability of processors and hard-drives [Stix 2001], provides a platform for future distributed storage systems to offer more services yet again. This paper covers a wide array of topics and challenges shaping Distributed Storage Systems today and beyond.

The rest of the paper is organised as follows: The next section (Section 2) discusses earlier works of relevance, serving to further complete and complement our work. In Section 3, we first introduce each of the topics covered throughout the taxonomy and then present a detailed taxonomy on distributed storage functionality, architecture, operating environment, usage patterns, autonomic management, federation, consistency and security. In Section 4, we provide a survey of representative storage systems. Rather than conducting an exhaustive survey, each system is selected based on its unique set of goals, so as to illustrate the topics covered in the taxonomy sections. Finally (Section 5), we conclude the paper with analysis of trends based on mapping of our taxonomy to representative systems and highlight outlook for future research work.

2. RELATED WORK

In this section we discuss studies which have investigated and surveyed distributed storage systems. The work of [Levy and Silberschatz 1990] provide an insight into the areas of replication, naming, caching and consistency and where applicable discusses solutions employed by surveyed systems. Whereas [Satyanarayanan 1989] not only provides a discussion of mechanisms employed by existing DSSs but also provides an insight into areas of research. A very comprehensive survey covering all distributed systems in general is presented by [Borghoff and Nast-Kolb 1989]. The survey begins by classifying distributed systems into two main categories, being DFSs and distributed operating systems and continues to survey each system based on a proposed standard template. The above surveys provide an insight into works from the late 70s right through to the early 90s. These papers serve to complement our survey and provide a comprehensive and thorough survey of early DSSs.

More recent works provide readers with an insight into Peer-to-Peer and Grid technologies. In their work [Milojicic et al. ; Oram 2001] discuss Peer-to-Peer technologies whilst covering a breadth of systems beyond distributed storage. Most recently [Hasan et al. 2005; Androutsellis-Theotokis and Spinellis 2004] provide a discussion of DSS with a particular focus on implementations of Peer-to-Peer routing overlays. Finally [Venugopal et al. 2006] covers the emerging field of Data Grids, focusing on access and the federation of globally distributed data, topics covered include replication, management and the processing of data.

The early works provide a great insight into issues relating to client-server filesystems, concepts which are essential to building a DSS today. The more recent surveys discuss cutting edge research, paying particular attention to Peer-to-Peer and Data Grid systems. Our paper encompasses distributed storage systems across the board, including Peer-to-Peer and Data Grid systems whilst classifying their functionality, architecture, operating environment and usage patterns. This paper aims to provide a birds eye view of current issues effecting research and development of distributed storage systems including routing, consistency, security, autonomic management and federation.

3. TOPIC INDEX

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

(1) *System Function (Section 3.1):* A classification of DSS functionality uncovers a wide array of behaviour, well beyond typical store and retrieve.

(2) *Storage Architecture (Section 3.2):* 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 3.3):* We identify various categories of operating environments and discuss how each influences design and architecture.

(4) *Usage Patterns (Section 3.4):* 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 3.5):* 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 3.6):* 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

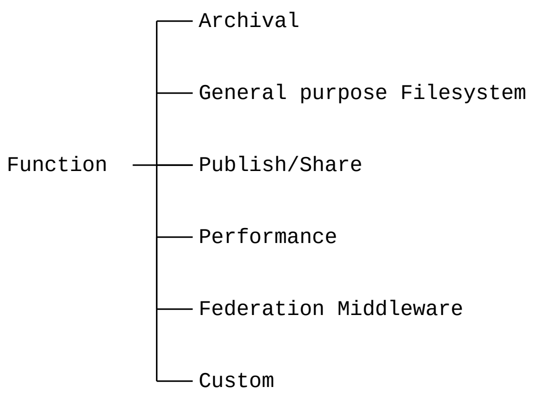

Fig. 1. system function taxonomy

traditional mechanisms as well as more recent approaches that have been developed for enforcing security in decentralised architectures.

(7) *Autonomic Management (Section 3.7):* Systems are increasing in complexity at an unsustainable rate. Research into autonomic computing [Kephart and Chess 2003] aims to overcome this dilemma by automating and abstracting away system complexity, simplifying maintenance and administration.

(8) *Federation (Section 3.8):* Many different formats and protocols are employed to store and access data, creating a difficult environment 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 3.9):* 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 client-server architectures to a dynamic and evolving approach as employed by Peer-to-Peer.

3.1 System Function

In this section we identify categories of distributed storage systems (Figure 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 [Druschel and Rowstron 2001] and CFS [Dabek et al. 2001].

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 [IEEE/ANSI Std. 1003.1] 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 by a POSIX compliant filesystem is a non-trivial matter, often met with compromises. Systems which fall into this category include NFS [Sandberg et al. 1985], Coda [Satyanarayanan 1990; Satyanarayanan et al. 1990], xFS [Anderson et al. 1996], Farsite [Adya et al. 2002] and Ivy [Muthitacharoen et al. 2002].

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 3.2). Examples of systems which fall into this category include Free Haven [Dingledine et al. 2000], Freenet [Clarke et al. 2001] and Publius [Waldman et al. 2000]. 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 [Oram 2001], inspired the subsequent development of other systems in this category; Gnutella [Oram 2001], MojoNation [Wilcox-O'Hearn 2002] and BitTorrent [Hasan et al. 2005] 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 [James V. Huber et al. 1995], Zebra [Hartman and Ousterhout 2001], PVFS [Carns et al. 2000], Lustre [Braam 2002], GPFS [Schmuck and Haskin 2002], Frangipani [Thekkath et al. 1997], PIOUS [Moyer and Sunderam 1994] and Galley [Nieuwejaar and Kotz 1996].

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

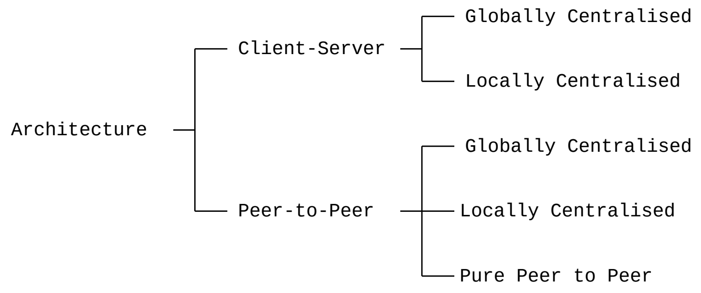

Fig. 2. architecture taxonomy

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 3.8. 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 [Chervenak et al. 2000; Hoschek et al. 2000; Venugopal et al. 2006; Baru et al. 1998] 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) [Ghemawat et al. 2003] and OceanStore [Kubiatowicz et al. 2000; Rhea et al. 2003], are examples of such systems. GFS has been built with a particular functional purpose which is reflected in its design (Section 4.7). 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 [Vazhkudai et al. 2005] 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.

3.2 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 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 [Schollmeier 2001]. 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 [Sandberg et al. 1985; Anderson et al. 1996; Morris et al. 1986; Satyanarayanan 1990; Thekkath et al. 1997; Vazhkudai et al. 2005; Ghemawat et al. 2003].

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 [Anderson et al. 1996; Satyanarayanan 1990; Thekkath et al. 1997; Vazhkudai et al. 2005; Ghemawat et al. 2003] 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 3.3). 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 [Oram 2001] 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 [Oram 2001] initially relied on broadcasting to relay queries although this proved to be a bottleneck, with as much as 50% [D. Dimitri 2002] of the traffic attributed to queries. To overcome this scalability bottleneck, a locally centralised architecture employs a 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 [Ding Choon-Hoong and Buyya 2005; Hasan et al. 2005], Clippee [Albrecht et al. 2003], Bittorrent [Hasan et al. 2005] and eDonkey [Tutschku 2004].

Without any central entities, a pure Peer-to-Peer architecture exhibits symmetrical 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

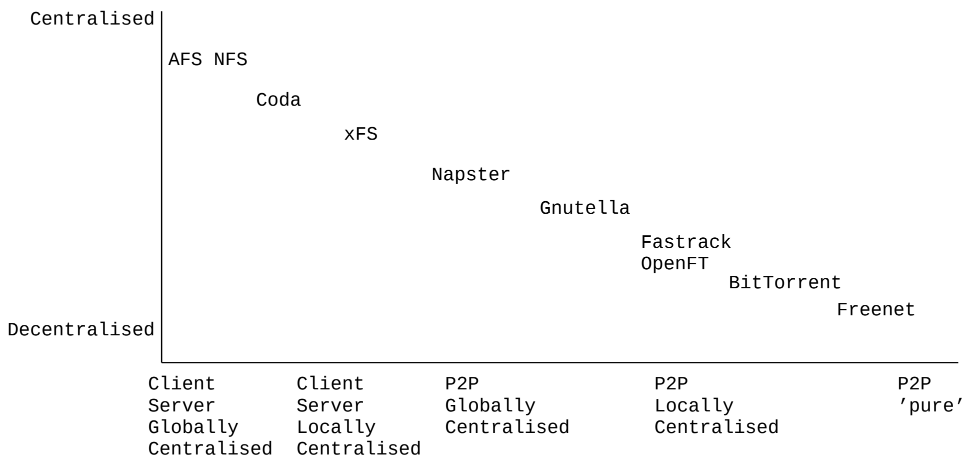

Fig. 3. architecture evolution

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 of an asymmetric [Oram 2001] 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 [Wilcox-O'Hearn 2002] 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 networking in a Peer-to-Peer system operation, the need for establishing trust and accountability becomes essential, which is 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 [Dingledine et al. 2000], Freenet [Clarke et al. 2001] and Ivy [Muthitacharoen et al. 2002].

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 unparallelled 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 3), adapting to challenges associated with operating across a dynamic global network.

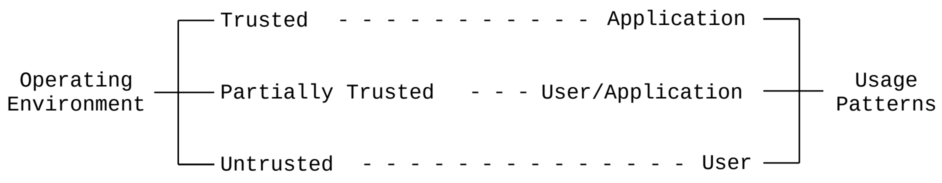

Fig. 4. operating environment and usage taxonomy

3.3 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 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 [IEEE/ANSI Std. 1003.1]. 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" [Shafer 2002; Rudis and Kostenbader 2003]. 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 [Oram 2001], a system can be subjected to a multitude of attacks [Dingledine 2000]. 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) [Wilcox-O'Hearn 2002], *tragedy of the commons* [Hardin 1968] and the *Slashdot effect* [Adler 1999].

Early research [Satyanarayanan 1992; Spasojevic and Satyanarayanan 1996] discusses issues associated with scaling up client-server distributed storage systems (Andrew [Morris et al. 1986] and Coda [Satyanarayanan 1990]) across a WAN. Some of the problems identified include (i) a lower level of trust existent 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 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.

3.4 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 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.

3.4.1 *Partially Trusted.* A study [Noble and Satyanarayanan 1994] focusing on the usage patterns of the Coda [Satyanarayanan 1990] (Section 4.4) storage system makes some interesting observations regarding file usage whilst disconnected from the file server. Coda employs an optimistic approach to consistency (Section 3.5) 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 [Douceur and Bolosky 1999] conducted across 4800 workstations within Microsoft found only half of the filesystem storage capacity to be utilised. These results inspired a subsequent feasibility study [Bolosky et al. 2000] 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 led to the development of Farsite [Adya et al. 2002] (Section 4.3).

A number of other empirical studies relevant to the partially trusted category include: A comparatively early study [Spasojevic and Satyanarayanan 1996] primarily focusing on the use of AFS [Howard et al. 1988] whilst another study adopted a developer's perspective [Gill et al. 1994], 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.

3.4.2 *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 [Wilcox-O'Hearn 2002] 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 [Feldman et al. 2003] resulting in Peer-to-Peer systems being vulnerable to free-riding, where users mainly consume services without providing any in return. Studies show that [Oram 2001; Feldman et al. 2003; Hughes et al. 2005] 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 [Hughes et al. 2005] found that 85% of users were free-riding. To discourage this and promote cooperation, the next generation of Peer-to-Peer systems (Maze [Yang et al. 2005], Bittorrent [Bittorrent]) provide incentives for users to contribute services.

3.4.3 *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 [Crandall et al. 1995] 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) [Ghemawat et al. 2003] (Section 4.7) 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 look up requests, (ii) the meta-data server had less chunk requests to process and consequently chunk entries to store and manage.

3.5 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 resultant 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.

3.5.1 *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 [Gray and Reuter 1993; Date 2002] 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 corrupted 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) [Haerder and Reuter 1983] 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 used 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 [Eswaran et al. 1976] 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 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 that 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*.

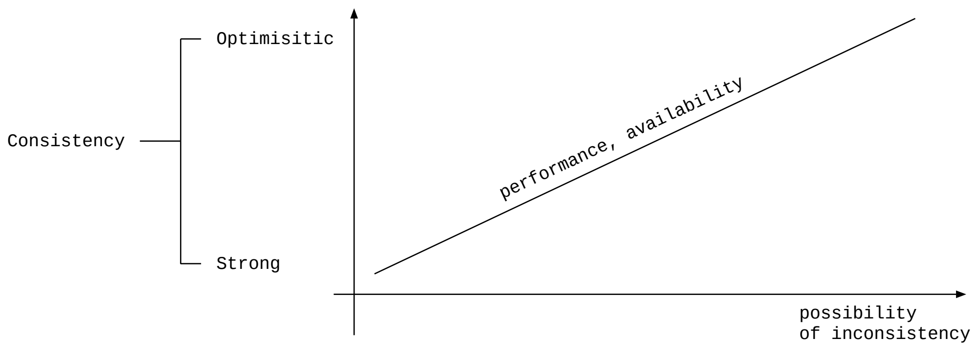

Fig. 5. strong vs optimistic consistency

(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.

3.5.2 *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 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 [Bernstein and Goodman 1983], 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 [Kung and Robinson 1981] which is also known as weak consistency. It is considered weak as it allows a 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.

3.5.3 *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 [Ries and Stonebraker 1977; 1979; Gray et al. 1994] has gone into applying locks and choosing an appropriate grain size. Choosing a large

Table I. strong consistency - impact on architecture and environment

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

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 [Thekkath et al. 1997] and OceanStore [Kubiatowicz et al. 2000]) 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 3.6). With these high overheads the choice to use a large block size is justified: e.g. 64MB used by the GFS [Ghemawat et al. 2003]. 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 I). 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.

3.5.4 *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 [Kistler and Satyanarayanan 1991; Gill et al. 1994] 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 [Anderson et al. 1996], Coda [Satyanarayanan 1990] and Ivy [Muthitacharoen et al. 2002]. Both Ivy and xFS employ a log structured filesystem, recording every filesystem operation into

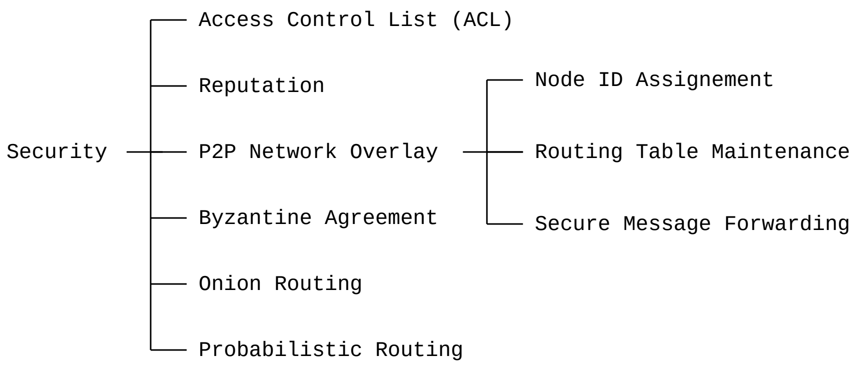

Fig. 6. security taxonomy

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 synchronise with the server's.

3.6 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 3.1), architecture (Section 3.2) and operating environment (Section 3.3) all have an impact on security and the various methods (Figure 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 administrative 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, 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 [Douceur 2002; Dingledine 2000]. 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 [Harrington and Jensen 2003]. 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 [Castro et al. 2002; Sit and Morris

2002]. Current Peer-to-Peer systems employ network overlays (Section 3.9) as their means to communicate and query other hosts. Securing a Peer-to-Peer network overlay [Castro et al. 2002] 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 [Clarke et al. 2001] 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 [Douceur 2002] attack. A centralised solution is proposed in [Castro et al. 2002], 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, [Castro et al. 2002] 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) [National Institute of Standards and Technology 1995] or Message Digest algorithm (MD5) [Rivest 1992]. 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 [Castro and Liskov 2000] enables the establishment of trust within an untrusted environment. The algorithm is based on a voting scheme, where

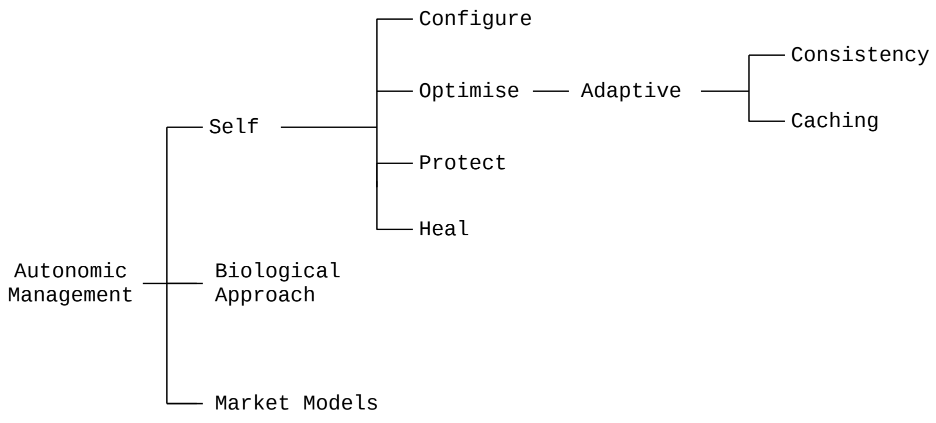

Fig. 7. autonomic management taxonomy

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 [Kubiatowicz et al. 2000] and Farsite [Adya et al. 2002] 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 [Dingledine et al. 2000] and MojoNation [Wilcox-O'Hearn 2002] use digital currency to encourage participating users to behave.

Systems such as Free Haven [Dingledine et al. 2000] and Freenet [Clarke et al. 2001] 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 [Dingledine et al. 2003; Marti and Garcia-Molina 2003] as this makes the system vulnerable to attacks. Onion routing [Oram 2001; Dingledine et al. 2004; Syverson et al. 1997] and probabilistic routing [Dingledine et al. 2000] are two methods employed to provide anonymous and censorship resistant communications medium.

3.7 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 components, increasing their complexity and the knowledge required to manage them. With this trend set to continue, research into addressing and managing complexity (Figure 7) has led to the emergence of autonomic computing [Horn 2001; Kephart and Chess 2003]. 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 [Barrett et al. 2003]. 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 [Horn 2001], was later distilled into four [Kephart and Chess 2003]; 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 II. Another approach to autonomic computing takes a more ad-hoc approach, drawing inspiration from biological models [Staab et al. 2003]. 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 [Ferguson et al. 1996; Wolski et al. 2001; Buyya 2002; Weglarz et al. 2004] 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 4.

(1) *Mungi [Heiser et al. 1998]:* is a Single-Address-Space Operating System (SASOS) which employs a commodity market model to manage storage quota.
(2) *Stanford Archival Repository Project [Cooper and Garcia-Molina 2002]:* apply a bartering mechanism, where institutions barter amongst each other for distributed storage services for the purpose of archiving and preserving information.
(3) *MojoNation [Wilcox-O'Hearn 2002]:* 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 [Kubiatowicz et al. 2000]:* is a globally scalable storage utility, providing paying users with a durable, highly available storage service by utilising untrusted infrastructure.
(5) *Storage Exchange [Placek and Buyya 2006]:* 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.

3.8 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 [Foster 2001]. Grid computing is faced with many challenges including: supporting cross domain administration, security, integration of heterogeneous systems, resource 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 identi-

Table II. autonomic computing and distributed storage

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 [Devarakonda et al. 2003] 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 [Kubiatowicz et al. 2000], and recently a more ad-hoc approach [Staab et al. 2003] inspired by the self-organising behaviour found in biological systems. The process of introspection is a structured three stage cyclical process: data is collected, analysed 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 reducing the load on the file server. Several efforts focusing on self-optimisation include GLOMAR [Cuce and Zaslavsky 2002], HAC [Castro et al. 1997] and a couple of proposals [Li et al. ; Li et al. 2005] 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 [Clarke et al. 2001] and Free Haven [Dingledine et al. 2000]) accommodate for a large array of attacks aimed to disrupt services and propose various methods to protect themselves.

fied across various scientific disciplines (Ecological [Jones 1998], High Energy Physics [Holtman 2001], Medicinal [Buyya 2001]). 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 [Chervenak et al. 2000] and SRB [Baru et al. 1998; Rajasekar et al. 2002] (Section 4.8) are examples of current research being carried out into federating storage services.

3.9 Routing and Network Overlays

The evolution of routing has evolved in step with distributed storage architecture. Early DSSs [Morris et al. 1986; Howard et al. 1988; Sandberg et al. 1985] 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 [Oram 2001] 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 [Oram 2001] 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 [D. Dimitri 2002]. 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 [Ding Choon-Hoong and Buyya 2005]). 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 [Plaxton et al. 1997; Zhao et al. 2003; Stoica et al. 2001; Rowstron and Druschel 2001; Ratnasamy et al. 2000; Dabek et al. 2003; Maymounkov and Mazieres 2002] 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 [Zhao et al. 2003; Rowstron and Druschel 2001]. 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 III) of all discussed routing algorithms, suggest that each has a varying capability regarding performance. Variables listed in Table III are described in detail in [Lua et al. 2005], which also provides a detailed description and comparison of network overlays.

Continuous research and development into network overlays has seen them evolve to

Table III. comparison of routing mechanisms

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	$O(log_b N)$
Pastry	Plaxton-style Global Mesh	$O(log_c N)$
Kademlia	X-OR based Look-up Mechanism	$O(log_e N)$

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 IV. routing and architecture taxonomy

	Centralised	Decentralised
Static	**1.** Client-Server NFS [Sandberg et al. 1985]	**2.** Replicated Servers xFS [Anderson et al. 1996], Coda [Satyanarayanan 1990]
Dynamic	**3.** Centralised Peer-to-Peer Napster [Oram 2001]	**4.** Peer-to-Peer Network Overlay Ivy [Muthitacharoen et al. 2002] OceanStore [Kubiatowicz et al. 2000]

support an increasing number of services. Some of these services include providing stronger consistency [Lynch et al. 2002], better query capability [Harren et al. 2002; Triantafillou and Pitoura 2003], anonymity [Freedman and Morris 2002] and censorship resistance [Hazel and Wiley 2002]. To consolidate the vast array of research, [Dabek et al. 2003] proposes a 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 IV). 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.

4. SURVEY

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 V lists all the surveyed systems tracing back to those characteristics discussed in the taxonomy.

4.1 OceanStore

OceanStore [Kubiatowicz et al. 2000] 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.

4.1.1 *Architecture.* OceanStore employs a 2-tier based architecture (Figure 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 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 [Castro and Liskov 2000] 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 [Bindel and Rhea 2000]. 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 [Demers et al. 1994]. 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 [Blömer et al. 1995] are employed to achieve redundancy. Erasure codes provide redundancy more efficiently then otherwise possible by replication [Weatherspoon et al. 2001].

The super-tier utilises Tapestry [Zhao et al. 2003], 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 3.9.

Data objects are stored (read-only), referenced by indirect blocks, in principle very much like a log structured filesystem [Rosenblum and Ousterhout 1992]. These indirect blocks

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)
StorageExchange	General purpose Filesystem	Globally Centralised	Untrusted	Strong	Static	POSIX	Large (institution)

Table V. distributed storage systems surveyed

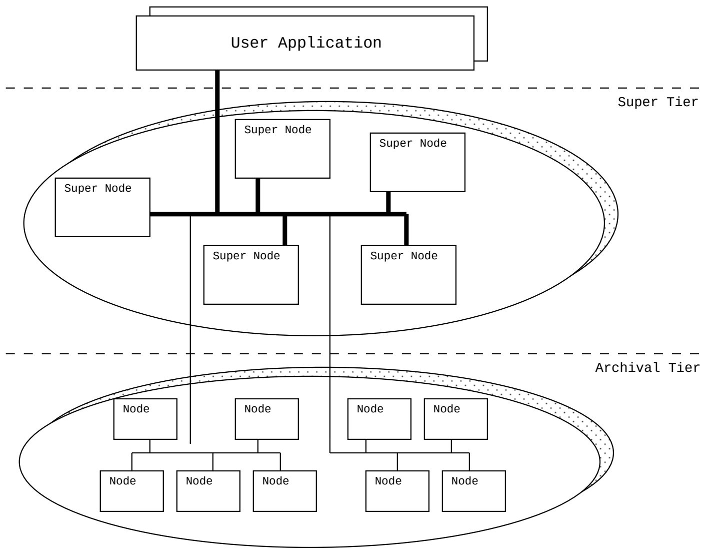

Fig. 8. OceanStore architecture

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.

4.1.2 *Implementation.* A prototype named Pond [Rhea et al. 2003] has been developed and released as open source under the BSD license and is available for download[1]. SEDA [Welsh et al. 2001] (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, 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.

4.1.3 *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

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

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 [Rhea et al. 2003] 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 [Eaton and Weis].

4.2 Free Haven

Free Haven [Dingledine et al. 2000] [Oram 2001] 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 [Oram 2001]) or users themselves (Gnutella [Oram 2001]) 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 [Dingledine 2000] provides a detailed classification 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.

4.2.1 *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 [Danezis et al. 2002], which provides an anonymous communications medium by utilising onion routing (Figure 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 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.

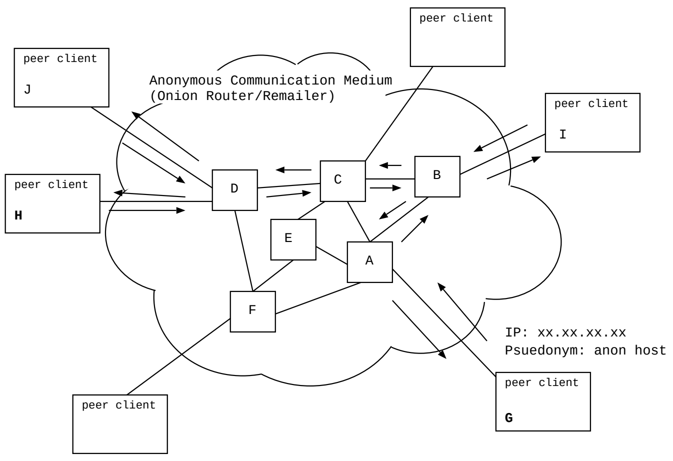

Fig. 9. Free Haven architecture

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 [Dingledine et al. 2003; Marti and Garcia-Molina 2003], 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 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.

4.2.2 *Implementation.* Free Haven has not been released, the website details the 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 [Dingledine et al. 2004], 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 anonymous re-mailer Mixminion [Danezis et al. 2002] and Tor [Dingledine et al. 2004] can be found on the Free Haven website.

4.2.3 *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 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 [Dingledine 2000] 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.

4.3 Farsite

The goal of Farsite [Adya et al. 2002] 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.

4.3.1 *Architecture*. The architecture is based on the following three concepts: client, directory group and a file host (Figure 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.

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

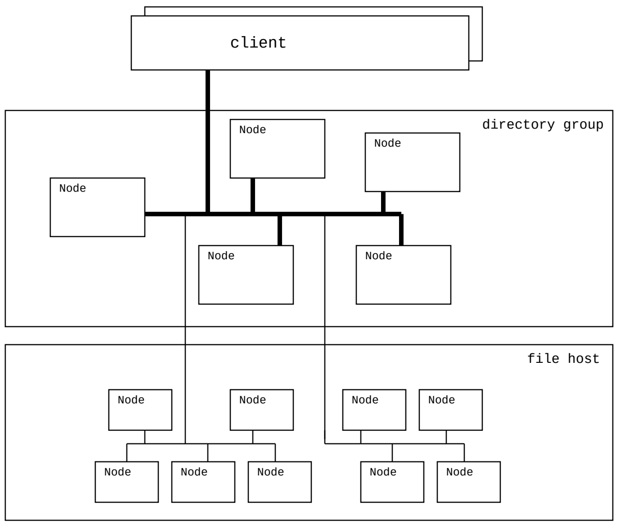

Fig. 10. Farsite architecture

They can also, as required, execute choirs and exhibit a degree of autonomic operation. The file host role consists of providing storage space for file data. We shall now discuss each role in greater detail.

The client provides an interface which emulates the behaviour 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 behaviour 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.

4.3.2 *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.

4.3.3 *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 an 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.

4.4 Coda

Coda [Satyanarayanan 1990; Satyanarayanan et al. 1990; Kistler and Satyanarayanan 1991] 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 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

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

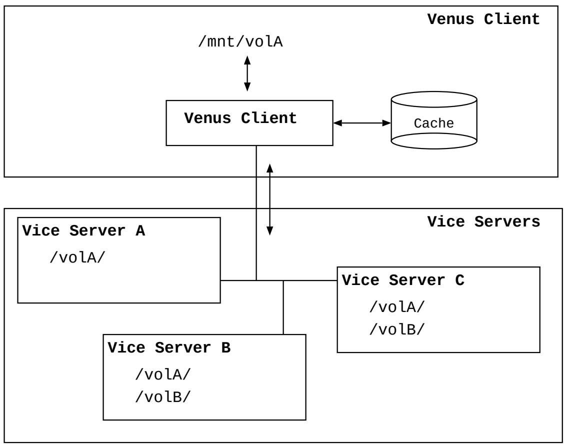

Fig. 11. Coda architecture

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.

4.4.1 *Architecture.* Coda is divided into two main components (Figure 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 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 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 keeps 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 [Kistler and Satyanarayanan 1991] 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.

4.4.2 *Implementation.* Coda was written in C and consists of two main components, the Vice server and the Venus client (Figure 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.

4.4.3 *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 [Kistler and Satyanarayanan 1991] 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.

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

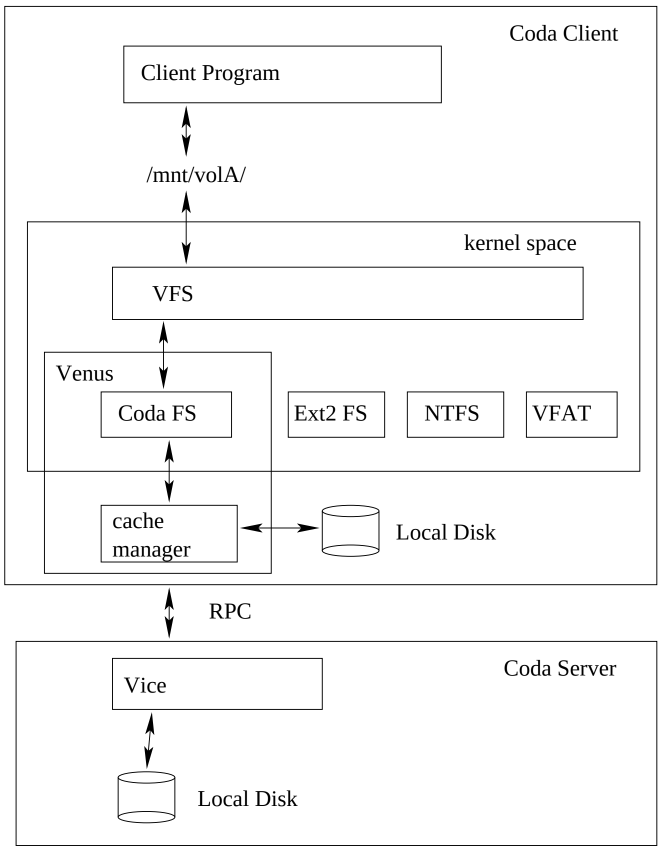

Fig. 12. Coda implementation architecture

4.5 Ivy

Ivy [Muthitacharoen et al. 2002] 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 [Oram 2001], Napster [Oram 2001]) 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 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.

4.5.1 *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 13). Chord/Dhash is used for providing a reliable Peer-to-Peer distributed storage mechanism. The Ivy server interfaces to Dhash, to

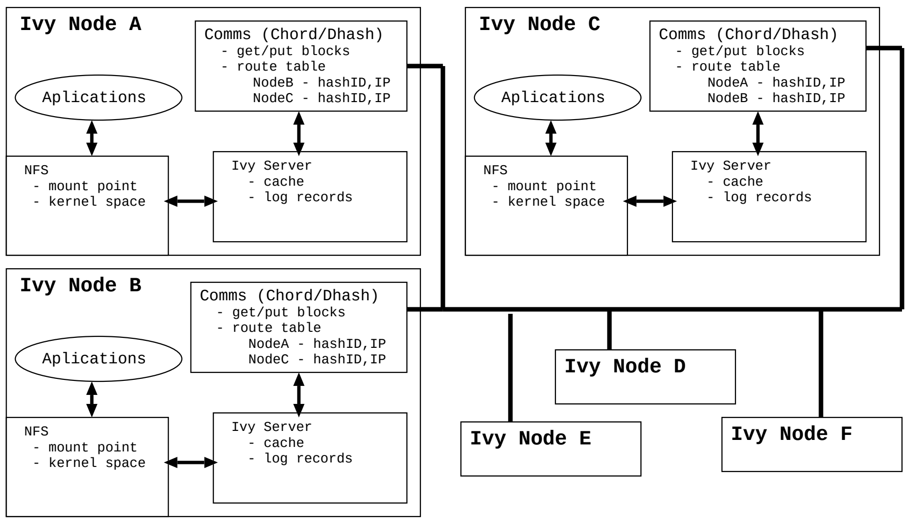

Fig. 13. Ivy architecture

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 [Stoica et al. 2001], a Peer-to-Peer network overlay (Section 3.9), 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 [Rosenblum and Ousterhout 1992]. 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 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.

4.5.2 *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.

4.5.3 *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 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 [Dabek et al. 2001], which, like Ivy, utilises Chord/DHash for distributing its storage but only supports a limited write-once/read-many interface.

4.6 Frangipani

Frangipani [Thekkath et al. 1997] 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.

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

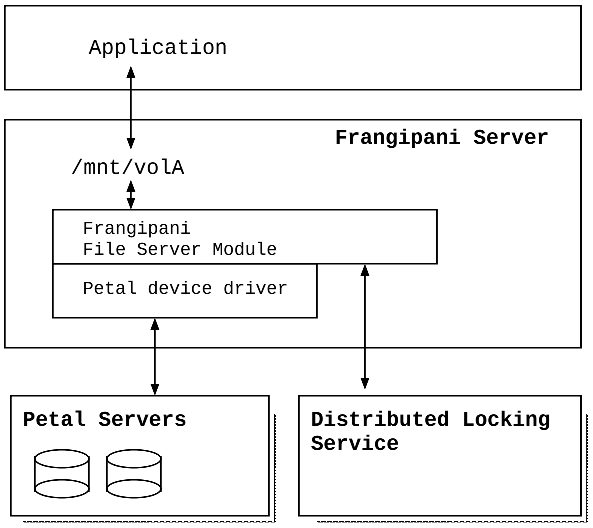

Fig. 14. Frangipani architecture

4.6.1 *Architecture.* Frangipani consists of the following main components: Petal Server, Distributed Locking Service and the Frangipani File Server Module (Figure 14). The Petal Server [Thekkath and Lee 1996] 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 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.

4.6.2 *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.

4.6.3 *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.

4.7 GFS

The Google File System [Ghemawat et al. 2003] 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 appended, 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.

4.7.1 *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

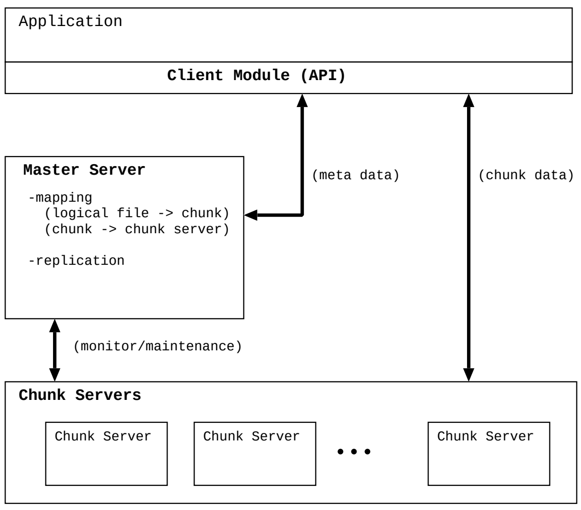

Fig. 15. GFS architecture

(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 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 hotspot, 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 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.

4.7.2 *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.

4.7.3 *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.

4.8 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)

[Baru et al. 1998; Rajasekar et al. 2002]. 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).

4.8.1 *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 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.

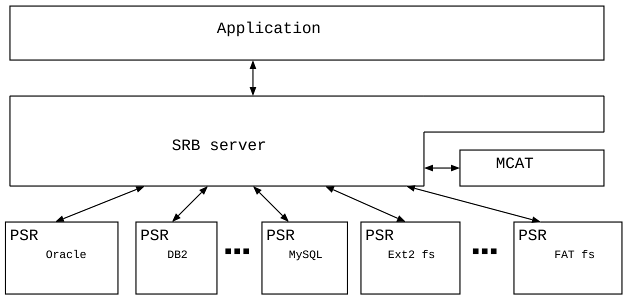

Fig. 16. SRB architecture

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 [Schroeder 1999], 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 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.

4.8.2 *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.

4.8.3 *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.

4.9 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 scien-

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

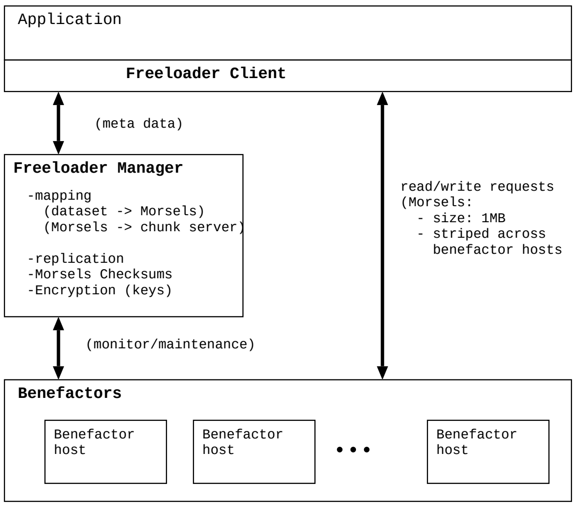

Fig. 17. Freeloader architecture

tists analyse the results. Freeloader [Vazhkudai et al. 2005] 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.

4.9.1 *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 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 [Ghemawat et al. 2003] and PVFS [Carns et al. 2000], 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 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.

4.9.2 *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.

4.9.3 *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

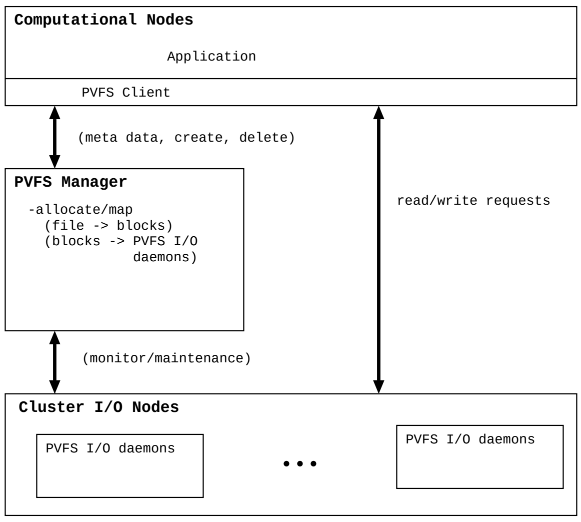

Fig. 18. PVFS architecture

expensive storage solutions like SANs.

4.10 PVFS

PVFS [Carns et al. 2000] 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.

4.10.1 *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 modify or recompile. The PVFS architecture is designed to operate as a client-server system (Figure 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 (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.

4.10.2 *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.

4.10.3 *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.

4.11 Storage Exchange

The Storage Exchange(SX) [Placek and Buyya 2006] is a platform allowing storage to be treated as a tradeable resource. Organisations with varying storage requirements can use the SX 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 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. Consumers and providers of storage use the storage exchange to advertise their requirements, which employs a Double Auction market model to efficiently allocate trades. Organisations may trade storage based on their current requirements, an organisation that is running low on storage it may purchase storage, alternatively if it finds that there is an abundance of storage it has the ability to lease out the excess storage.

4.11.1 *Architecture.* The storage exchange employs are hierarchical architecture (Figure 19) consisting of the following four main components: (i) Storage Client: provides an interface for the user to access storage services, (ii) Storage Provider: harnesses avail-

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

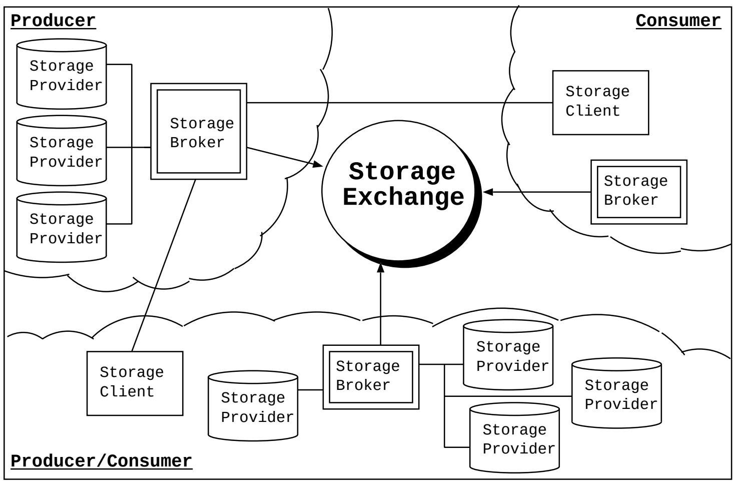

Fig. 19. storage exchange architecture

able storage on installed host whilst servicing requests from Storage Client, (iii) Storage Broker: manages inhouse storage capacity and trades storage based upon storage service requirements of institution, and finally the (iv) Storage Exchange: a trading platform used by Storage Brokers to trade storage. We shall now discuss each of the components in greater detail:

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 can not be installed on servers or dedicated hosts. The Storage Provider is responsible for keeping the organisations broker up-to-date with various usage statistics and service incoming storage requests initiated by Storage Clients.

Storage Client: An organisation wishing to utilise a negotiated storage contract will need to use a Storage Client. A user will configure the Storage Client with the storage contract details. The Storage Client then uses these details to authenticate itself with the provider's Storage Broker and upon successful authentication the Storage Client requests a mount for the volume. The provider's Storage Broker then looks up the Storage Providers responsible for servicing the storage contract and instructs them to connect to the Storage Client. Once the Storage Providers establish a connection with the Storage Client, the Storage Client then provides a filesystem like interface, much like an NFS[Sandberg et al. 1985] 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 to be linked with special libraries.

Storage Broker: For an organisation to be able to participate in the SX platform they will need to use a Storage Broker. The Storage Broker enables the organisation to trade and utilise storage services from other organisations. The Storage Broker needs to be configured to reflect how it should best serve the organisations interests. From a consumer's perspective the Storage Broker will need to know the organisations 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 will contact the storage exchange with its requirements.

Storage Exchange (SX): The Storage Exchange component provides a platform for Storage Brokers to advertise their storage services and requirements. The SX is a trusted entity responsible for executing a market model and determining how storage services are traded. When requests for storage are allocated to available storage services the Storage Exchange generates a storage contract. The storage contract contains a configuration of the storage policy forming a contract binding the provider to fulfil the service at the determined price. In a situation where either the provider or consumer breaches a storage contract, the SX will keep a record of reputation for each organisation which can be used to influence future trade allocations.

4.11.2 *Implementation.* The storage provider and storage client components have been written in C. The storage client utilises the FUSE library [FUSE 2000] to provide a local mount point of the storage volume in user space. The Storage Broker and Storage Exchange have both been written in Java. Storage Providers support replication, allowing volumes to be replicated across multiple storage providers, ensuring better reliability and availability. Communication between components is carried out via TCP socket communication. The storage exchange accepts offers from storage brokers and employs a clearing algorithm to allocate trades. Performance evaluation provided [Placek and Buyya 2006] focuses on the storage exchange and comparing the different clearing algorithms it employs.

4.11.3 *Summary.* The SX platform provides organisations with various storage services and requirements the capability to trade and exchange these services. The platform aims to federate storage services, allowing organisations to find storage services which better meet their requirements whilst better utilising their available infrastructure. Organisations are able to scavenge storage services across their network of workstations and with the use of the SX platform lease it out globally. The Storage Exchange has much scope for future research, laying a foundation for further investigation into utilising economic principles to achieve autonomic management [Pattnaik et al. 2003] of storage services.

5. CONCLUSION

This paper discusses a wide range of topics and areas of research relevant to distributed storage systems. We begin by proposing an abstract model, which gives an overall view, demonstrating how each topic fits into the big scheme of things. The abstract model is subsequently used as a road map for the work discussed throughout the taxonomy sections. The taxonomy has two main sections to it, one section covers *System Wide* topics the other *Core*. *System-Wide* covers storage functionality, architecture, operating environment and usage patterns. Topics included in the *Core* section focus on autonomic storage, Federation, Consistency, Security and finally Routing and Network Overlays. The taxonomy sections are followed by the survey, which covers a wide range of systems, from systems which provide storage utility on a global scale (OceanStore) to systems which provide high level of accessibility to mobile users (Coda). Each system has been selected for its unique properties serving to exemplify topics discussed through our taxonomy section. Table V demonstrates how each surveyed system traces back to our taxonomy and abstract model.

Throughout our investigation we observe various relationships amongst topics covered. In the taxonomy we see that distributed storage functionality, architecture, operating environment, usage patterns, routing and consistency all share various points of dependencies. To illustrate, we observe that a system providing strong consistency requires the underlying architecture to have some level of central control and predictability such that strong consistency can be enforced. In contrast, systems which adopt an architecture with little or no centralisation, may only support an optimistic approach to consistency or avoid dealing with issues of consistency altogether by supporting a WORM usage pattern. These class of systems typically provide publishing capability scaling to a dynamic global audience and as such employ an architecture with limited centralisation to suite.

Whilst these are some of the issues facing current development of DSSs, there are many emerging challenges on the horizon, two challenges include increasing in complexity [Staab et al. 2003; Kephart and Chess 2003; Horn 2001] and global federation of resources [Foster 2001; Venugopal et al. 2006; Rajasekar et al. 2002] which have given rise to autonomic computing and Grid computing respectively. Autonomic computing aims to overcome the burden imposed by complexity through abstracting it away from users and developers. Whereas Grid computing allows institutions to collaborate and share resources across geographic and administrative boundaries. Whilst these are emerging areas of research, the more established issues including consistency, routing and security, are no less important. Hence they will continue to evolve and serve a key role in the development of Distributed Storage Systems.

Future research and development of DSS is very much dependent on the underlying network infrastructure. To illustrate, the advent of the Internet saw a subsequent flurry of research aiming to harness new possibilities which come with global connectivity. With networking predicted to undergo another quantum leap, outpacing development of computational and storage [Stix 2001] devices, it is certain that this radical change will result in yet another wave of research, seeking to better utilise the advances in network infrastructure. Laying the foundation for exciting and innovative research into Distributed Storage Systems.

REFERENCES

ADLER, S. 1999. The slashdot effect – an analysis of three Internet publications.

ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. 2002. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Operating Systems Review 36,* SI, 1–14.

ALBRECHT, K., ARNOLD, R., AND WATTENHOFER, R. 2003. Clippee: A large-scale client/peer system.

ANDERSON, T., DAHLIN, M., NEEFE, J., PATTERSON, D., ROSELLI, D., AND WANG, R. 1996. Serverless Network File Systems. *ACM Transactions on Computer Systems 14,* 1 (Feb.), 41–79.

ANDROUTSELLIS-THEOTOKIS, S. AND SPINELLIS, D. 2004. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys 36,* 4, 335–371.

BARRETT, R., CHEN, Y.-Y. M., AND MAGLIO, P. P. 2003. 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*. ACM Press, New York, NY, USA, 1068–1069.

BARU, C., MOORE, R., RAJASEKAR, A., AND WAN, M. 1998. The sdsc storage resource broker. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 5.

BERNSTEIN, P. A. AND GOODMAN, N. 1983. The failure and recovery problem for replicated databases. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*. ACM Press, New York, NY, USA, 114–122.

BINDEL, D. AND RHEA, S. 2000. The design of the oceanstore consistency mechanism.

BITTORRENT. http://www.bittorrent.com/protocol.html.

BLÖMER, J., KALFANE, M., KARP, R., KARPINSKI, M., LUBY, M., AND ZUCKERMAN, D. 1995. An xor-based erasure-resilient coding scheme. Tech. Rep. TR-95-048, International Computer Science Institute, Berkeley, USA, Berkley.

BOLOSKY, W. J., DOUCEUR, J. R., ELY, D., AND THEIMER, M. 2000. 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*. ACM Press, New York, NY, USA, 34–43.

BORGHOFF, U. M. AND NAST-KOLB, K. 1989. Distributed systems: A comprehensive survey. Tech. Rep. TUM-I8909, Postfach 20 24 20, D-8000 München 2, Germany.

BRAAM, P. J. 2002. The lustre storage architecture. Cluster File Systems Inc. Architecture, design, and manual for Lustre. http://www.lustre.org/docs/lustre.pdf.

BUYYA, R. 2001. The virtual laboratory project: Molecular modeling for drug design on grid. In *IEEE Distributed Systems Online*.

BUYYA, R. 2002. Economic-based distributed resource management and scheduling for grid computing. Ph.D. thesis, Monash University, Melbourne, Australia.

CARNS, P. H., LIGON III, W. B., ROSS, R. B., AND THAKUR, R. 2000. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*. USENIX Association, Atlanta, GA, 317–327.

CASTRO, M., ADYA, A., LISKOV, B., AND MYERS, A. C. 1997. HAC: Hybrid adaptive caching for distributed storage systems. In *ACM Symposium on Operating Systems Principles (SOSP)*. Saint Malo, France, 102–115.

CASTRO, M., DRUSHEL, P., GANESH, A., ROWSTRON, A., AND WALLACH, D. 2002. Secure routing for structured peer-to-peer overlay networks.

CASTRO, M. AND LISKOV, B. 2000. Proactive recovery in a Byzantine-Fault-Tolerant system. In *In Proc. of Sigmetrics*. 273–288.

CHERVENAK, A., FOSTER, I., KESSELMAN, C., SALISBURY, C., AND TUECKE, S. 2000. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. In *Journal of Network and Computer Applications*. Vol. 23.

CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. 2001. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science 2009*, 46+.

COOPER, B. F. AND GARCIA-MOLINA, H. 2002. Peer-to-peer data trading to preserve information. *ACM Transactions on Information Systems 20,* 2, 133–170.

CRANDALL, P. E., AYDT, R. A., CHIEN, A. A., AND REED, D. A. 1995. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*. IEEE Computer Society Press, San Diego, CA.

CUCE, S. AND ZASLAVSKY, A. B. 2002. Adaptable consistency control mechanism for a mobility enabled file system. In *MDM '02: Proceedings of the Third International Conference on Mobile Data Management*. IEEE Computer Society, Washington, DC, USA, 27–34.

D. DIMITRI, G. ANTONIO, K. B. 2002. Analysis of peer-to-peer network security using gnutella.

DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. 2001. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*. Chateau Lake Louise, Banff, Canada.

DABEK, F., ZHAO, B., DRUSCHEL, P., AND STOICA, I. 2003. Towards a common api for structured peer-to-peer overlays.

DANEZIS, G., DINGLEDINE, R., AND MATHEWSON, N. 2002. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*. 2–15.

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

DEMERS, A. J., PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND WELCH, B. B. 1994. The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*. Santa Cruz, California, 2–7.

DEVARAKONDA, M., SEGAL, A., AND CHESS, D. 2003. 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*. IEEE Computer Society, Washington, DC, USA, 89.

DING CHOON-HOONG, S. N. AND BUYYA, R. 2005. Peer-to-peer networks for content sharing. In *Peer-to-Peer Computing: Evolution of a Disruptive Technology*, R. Subramanian and B. Goodman, Eds. Idea Group Publishing, Hershey, PA, USA, 28–65.

DINGLEDINE, R. 2000. The free haven project: Design and deployment of an anonymous secure data haven. Master's thesis, MIT.

DINGLEDINE, R., FREEDMAN, M. J., AND MOLNAR, D. 2000. The free haven project: Distributed anonymous storage service. In *Workshop on Design Issues in Anonymity and Unobservability*. Number 2009 in LNCS. 67–95.

DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. 2003. Reputation in p2p anonymity systems.

DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. 2004. Tor: The Second-Generation Onion Router. In *Proceedings of the Seventh USENIX Security Symposium*.

DOUCEUR, J. 2002. The sybil attack.

DOUCEUR, J. R. AND BOLOSKY, W. J. 1999. A large-scale study of file-system contents. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM Press, 59–70.

DRUSCHEL, P. AND ROWSTRON, A. 2001. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*. Schloss Elmau, Germany, 75–80.

EATON, P. AND WEIS, S. Examining the security of a file system interface to oceanstore.

ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. 1976. The notions of consistency and predicate locks in a database system. *Communications of the ACM 19,* 11, 624–633.

FELDMAN, M., LAI, K., CHUANG, J., AND STOICA, I. 2003. Quantifying disincentives in peer-to-peer networks. In *1st Workshop on Economics of Peer-to-Peer Systems*.

FERGUSON, D. F., NIKOLAOU, C., SAIRAMESH, J., AND YEMINI, Y. 1996. Economic models for allocating resources in computer systems. 156–183.

FOSTER, I. T. 2001. 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*. Springer-Verlag, London, UK, 1–4.

FREEDMAN, M. J. AND MORRIS, R. 2002. Tarzan: a peer-to-peer anonymizing network layer. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*. ACM Press, New York, NY, USA, 193–206.

FUSE. 2000. http://sourceforge.net/projects/fuse/.

GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. 2003. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM Press, 29–43.

GILL, D. S., ZHOU, S., AND SANDHU, H. S. 1994. 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*. ACM Press, 276–277.

GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

GRAY, J. N., LORIE, R. A., PUTZOLU, G. R., AND TRAIGER, I. L. 1994. Granularity of locks and degrees of consistency in a shared data base. 181–208.

HAERDER, T. AND REUTER, A. 1983. Principles of transaction-oriented database recovery. *ACM Computing Survey 15,* 4, 287–317.

HARDIN, G. 1968. Tragedy of the commons. *Science 162,* 3859, 1243–1248.

HARREN, M., HELLERSTEIN, J., HUEBSCH, R., LOO, B., SHENKER, S., AND STOICA, I. 2002. Complex queries in dht-based peer-to-peer networks.

HARRINGTON, A. AND JENSEN, C. 2003. Cryptographic access control in a distributed file system. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*. ACM Press, New York, NY, USA, 158–165.

HARTMAN, J. H. AND OUSTERHOUT, J. K. 2001. The Zebra striped network file system. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, H. Jin, T. Cortes, and R. Buyya, Eds. IEEE Computer Society Press and Wiley, New York, NY, 309–329.

HASAN, R., ANWAR, Z., YURCIK, W., BRUMBAUGH, L., AND CAMPBELL, R. 2005. A survey of peer-to-peer storage techniques for distributed file systems. In *IEEE International Conference on Information Technology (ITCC)*. IEEE.

HAZEL, S. AND WILEY, B. 2002. Achord: A variant of the chord lookup service for use in censorship resistant peer-to-peer publishing systems.

HEISER, G., ELPHINSTONE, K., VOCHTELOO, J., RUSSELL, S., AND LIEDTKE, J. 1998. The Mungi single-address-space operating system. *Software Practice and Experience 28,* 9, 901–928.

HOLTMAN, K. 2001. Cms data grid system overview and requirements.

HORN, P. 2001. Autonomic computing: Ibm's perspective on the state of information technology.

HOSCHEK, W., JAEN-MARTINEZ, F. J., SAMAR, A., STOCKINGER, H., AND STOCKINGER, K. 2000. Data Management in an International Data Grid Project. In *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (GRID '00)*. Springer-Verlag, London, UK, Bangalore, India.

HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. 1988. Scale and performance in a distributed file system. *ACM Transactions Computing Systems 6,* 1, 51–81.

HUGHES, D., COULSON, G., AND J.WALKERDINE. 2005. Freeriding on gnutella revisited: the bell tolls? In *IEEE Distributed Systems Online*.

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

JAMES V. HUBER, J., CHIEN, A. A., ELFORD, C. L., BLUMENTHAL, D. S., AND REED, D. A. 1995. Ppfs: a high performance portable parallel file system. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*. ACM Press, New York, NY, USA, 385–394.

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

KEPHART, J. O. AND CHESS, D. M. 2003. The vision of autonomic computing. *Computer 36,* 1, 41–50.

KISTLER, J. J. AND SATYANARAYANAN, M. 1991. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*. Vol. 25. ACM Press, Asilomar Conference Center, Pacific Grove, U.S., 213–225.

KUBIATOWICZ, J., BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. 2000. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM.

KUNG, H. T. AND ROBINSON, J. T. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 6,* 2, 213–226.

LEVY, E. AND SILBERSCHATZ, A. 1990. Distributed file systems: concepts and examples. *ACM Computing Survey 22,* 4, 321–374.

LI, Z., CHEN, Z., AND ZHOU, Y. 2005. Mining block correlations to improve storage performance. *Transactions on Storage 1,* 2, 213–245.

LI, Z., SRINIVASAN, S. M., CHEN, Z., ZHOU, Y., TZVETKOV, P., YAN, X., AND HAN, J. Using data mining for discovering patterns in autonomic storage systems.

LUA, K., CROWCROFT, J., PIAS, M., SHARMA, R., AND LIM, S. 2005. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, 72–93.

LYNCH, N. A., MALKHI, D., AND RATAJCZAK, D. 2002. Atomic data access in distributed hash tables. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*. Springer-Verlag, London, UK, 295–305.

MARTI, S. AND GARCIA-MOLINA, H. 2003. Identity crisis: Anonymity vs. reputation in p2p systems. In *Peer-to-Peer Computing*. IEEE Computer Society, 134–141.

MAYMOUNKOV, P. AND MAZIERES, D. 2002. 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*. Springer-Verlag, London, UK, 53–65.

MILOJICIC, D. S., KALOGERAKI, V., LUKOSE, R., NAGARAJA, K., PRUYNE, J., RICHARD, B., ROLLINS, S., AND XU, Z. Peer-to-peer computing.

MORRIS, J. H., SATYANARAYANAN, M., CONNER, M. H., HOWARD, J. H., ROSENTHAL, D. S., AND SMITH, F. D. 1986. Andrew: a distributed personal computing environment. *Communications of the ACM 29,* 3, 184–201.

MOYER, S. A. AND SUNDERAM, V. S. 1994. PIOUS: A scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*. 71–78.

MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. 2002. Ivy: A read/write peer-to-peer file system. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*. USENIX.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. 1995. *FIPS PUB 180-1: Secure Hash Standard*. Supersedes FIPS PUB 180 1993 May 11.

NIEUWEJAAR, N. AND KOTZ, D. 1996. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*. ACM Press, Philadelphia, PA, 374–381.

NOBLE, B. D. AND SATYANARAYANAN, M. 1994. *An empirical study of a highly available file system*. New York, NY, USA.

ORAM, A. 2001. *Peer-to-Peer : Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Sebastopol, CA.

PATTNAIK, P., EKANADHAM, K., AND JANN, J. 2003. *Grid Computing: Making the Global Infrastructure a Reality*. Wiley Press, New York, NY, USA, Chapter Autonomic Computing and GRID.

PLACEK, M. AND BUYYA, R. 2006. Storage exchange: A global trading platform for storage services. In *12th International European Parallel Computing Conference (EuroPar)*. LNCS. Springer-Verlag, Berlin, Germany, Dresden, Germany.

PLAXTON, C. G., RAJARAMAN, R., AND RICHA, A. W. 1997. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*. 311–320.

RAJASEKAR, A., WAN, M., AND MOORE, R. 2002. Mysrb & srb: Components of a data grid.

RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. 2000. A scalable content addressable network. Tech. Rep. TR-00-010, Berkeley, CA.

REED, D. A., MENDES, C. L., DA LU, C., FOSTER, I., AND KESSELMAN, C. 2003. *The Grid 2: Blueprint for a New Computing Infrastructure - Application Tuning and Adaptation*, Second ed. Morgan Kaufman, San Francisco, CA. pp.513-532.

RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. 2003. Pond: The oceanstore prototype. In *Proceedings of the Conference on File and Storage Technologies*. USENIX.

RIES, D. R. AND STONEBRAKER, M. 1977. Effects of locking granularity in a database management system. *ACM Transactions on Database Systems 2,* 3, 233–246.

RIES, D. R. AND STONEBRAKER, M. R. 1979. Locking granularity revisited. *ACM Transactions on Database Systems 4,* 2, 210–227.

RIVEST, R. L. 1992. The MD5 Message Digest Algorithm. RFC 1321.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10,* 1, 26–52.

ROWSTRON, A. AND DRUSCHEL, P. 2001. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In IFIP/ACM International Conference on Distributed Systems Platforms (Middleware). *Lecture Notes in Computer Science 2218*, 329–350.

RUDIS, B. AND KOSTENBADER, P. 2003. The enemy within: Firewalls and backdoors.

SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. 1985. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.* Portland OR (USA), 119–130.

SATYANARAYANAN, M. 1989. A survey of distributed file systems. Tech. Rep. CMU-CS-89-116, Pittsburgh, Pennsylvania.

SATYANARAYANAN, M. 1990. Scalable, secure, and highly available distributed file access. *Computer 23,* 5, 9–18, 20–21.

SATYANARAYANAN, M. 1992. The influence of scale on distributed file system design. *IEEE Transactions on Software Engineering 18,* 1, 1–8.

SATYANARAYANAN, M., KISTLER, J. J., KUMAR, P., OKASAKI, M. E., SIEGEL, E. H., AND STEERE, D. C. 1990. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers 39,* 4, 447–459.

SCHMUCK, F. AND HASKIN, R. 2002. GPFS: A shared-disk file system for large computing clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*. 231–244.

SCHOLLMEIER, R. 2001. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Peer-to-Peer Computing*. IEEE Computer Society, 101–102.

SCHROEDER, W. 1999. The sdsc encryption/authentication (sea) system. *Concurrency - Practice and Experience 11,* 15, 913–931.

SHAFER, S. T. 2002. Corporate espionage the enemy within. Red Herring.

SIT, E. AND MORRIS, R. 2002. Security considerations for peer-to-peer distributed hash tables.

SPASOJEVIC, M. AND SATYANARAYANAN, M. 1996. An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems 14,* 2, 200–222.

STAAB, S., HEYLIGHEN, F., GERSHENSON, C., FLAKE, G. W., PENNOCK, D. M., FAIN, D. C., ROURE, D. D., ABERER, K., SHEN, W.-M., DOUSSE, O., AND THIRAN, P. 2003. Neurons, viscose fluids, freshwater polyp hydra-and self-organizing information systems. *IEEE Intelligent Systems 18,* 4, 72–86.

STIX, G. 2001. The ultimate optical networks: The triumph of the light. *Scientific American 284,* 1, 80–86.

STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. 2001. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*. 149–160.

SYVERSON, P. F., GOLDSCHLAG, D. M., AND REED, M. G. 1997. Anonymous connections and onion routing. In *IEEE Symposium on Security and Privacy*. Oakland, California, 44–54.

THEKKATH, C. A. AND LEE, E. K. 1996. Petal: Distributed virtual disks. In *Proc. 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 84–92.

THEKKATH, C. A., MANN, T., AND LEE, E. K. 1997. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*. 224–237.

TRIANTAFILLOU, P. AND PITOURA, T. 2003. Towards a unifying framework for complex query processing over structured peer-to-peer data networks. In *DBISP2P*, K. Aberer, V. Kalogeraki, and M. Koubarakis, Eds. Lecture Notes in Computer Science, vol. 2944. Springer, 169–183.

TUTSCHKU, K. 2004. A measurement-based traffic profile of the edonkey filesharing service. In *PAM*, C. Barakat and I. Pratt, Eds. Lecture Notes in Computer Science, vol. 3015. Springer, 12–21.

VAZHKUDAI, S. S., MA, X., FREEH, V. W., TAMMINEEDI, J. W. S. N., , AND SCOTT., S. L. 2005. Freeloader: Scavenging desktop storage resources for scientific data. In *IEEE/ACM Supercomputing 2005 (SC|05)*. IEEE Computer Society, Seattle, WA.

VENUGOPAL, S., BUYYA, R., AND RAMAMOHANARAO, K. 2006. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Computing Survey 28*.

WALDMAN, M., RUBIN, A., AND CRANOR, L. 2000. Publiues: a robust tamper-evident censorship-resistant web publishing system. In *Proceedings of the Nineth USENIX Security Symposium*. USENIX Association, Denver, CO, USA.

WEATHERSPOON, H., WELLS, C., EATON, P., ZHAO, B., AND KUBIATOWICZ, J. 2001. Silverback: A global-scale archival system.

WEGLARZ, J., NABRZYSKI, J., AND SCHOPF, J., Eds. 2004. *Grid resource management: state of the art and future trends*. Kluwer Academic Publishers, Norwell, MA, USA.

WELSH, M., CULLER, D., AND BREWER, E. 2001. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*. ACM Press, New York, NY, USA, 230–243.

WILCOX-O'HEARN, B. 2002. Experiences deploying a large-scale emergent network. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. Springer-Verlag, 104–110.

WOLSKI, R., PLANK, J. S., BREVIK, J., AND BRYAN, T. 2001. G-commerce: Market formulations controlling resource allocation on the computational grid. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEEE, San Francisco.

YANG, M., ZHANG, Z., LI, X., AND DAI, Y. 2005. 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.

ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. 2003. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*. Special Issue on Service Overlay Networks, to appear.