ABSTRACT

TAMMINEEDI, NANDAN Design of the management component in a scavenged storage environment. (Under the direction of Assistant Professor Dr. Xiaosong Ma).

High-end mass storage systems are increasingly becoming popular in supercomputing facilities for their large storage capacities and superior data delivery rates. End-users, on the other hand, face problems in processing this data on their local machines due to the limited disk bandwidth and memory. The Freeloader project is based on the premise that in a LAN environment, a large number of such workstations collectively represent significant storage space and aggregate I/O bandwidth, if harnessed when idle. Aggregation of these precious resources is made viable by the high speed interconnect that exists between nodes in a LAN. The FreeLoader project is an effort to aggregate free storage space, and I/O bandwidth contributions from commodity desktops to provide a shared cache/scratch space for large, immutable data sets. Striping is initially used to distribute data among multiple workstations, and this enables subsequent retrieval of data in the form of parallel streams from multiple workstations.

In this thesis, we present the management component of the Freeloader project. We discuss the functionality of the management component in terms of data placement and maintenance of information about workstations which donate storage. We show how the striping of data maximizes retrieval rates and helps in load balancing. We present the choices we face in the design of the management component and how to minimize its overheads. We also model the entire Freeloader cloud as a cache space with an eviction policy, due to the dynamic nature of space contributions and the limited amount of donated space. We discuss how the management component handles data set eviction in a manner that exploits temporal locality based on a history of accesses. We also discuss experimental results which show the impact of different striping parameters on the data access rates, and the viability of Freeloader in comparison to traditional data retrieval from high-end storage systems.

Design of the management component in a scavenged storage environment

by

Nandan Tammineedi

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

Raleigh

2005

Approved By:

Dr. Khaled Harfoush	Dr. Vincent Freeh

Dr. Xiaosong Ma
Chair of Advisory Committee

Biography

Nandan Tammineedi was born on 27^{th} August 1981. He is from Bangalore, India, and he received his Bachelor of Engineering in Information Science from Vishweshwaraiah Technological University located in the same city, in 2003. He started his graduate studies at North Carolina State University in Fall 2003. With the defense of this thesis, he receives the Master of Science degree in Computer Science from NCSU in May 2005.

Acknowledgements

I would like to thank Dr. Xiaosong Ma for her guidance, support and faith throughout the project. I thank Dr. Sudharshan Vazhkudai, my mentor during my summer internship at ORNL, for his guidance and insight during the initial days of the project, and for his continuing role in its development. I am also grateful to Dr. Vincent Freeh for his constant promotion of discussion and his role in the project. I would also like to thank Dr. Khaled Harfoush for agreeing to serve on my thesis committee. I also thank Jonathan Strickland, with whom numerous hours were spent developing Freeloader code. His work has laid the foundation for the project and his code-writing skills are admirable. Special thanks to Nandini Kappiah, who supported me through my thesis and much more.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Scientific discoveries are increasingly dependant on analysis of massive, high-resolution data, produced by observatory instruments or computer simulations [15]. As a result, storage area clusters, data centers and high-end mass storage systems are becoming popular in supercomputing facilities due to their large storage capacities and their ability to deliver this stored data at high rates. While these systems serve the data needs for a data-intensive application, end-users of scientific data are often limited by their local resources. Local data storage is important for the "last mile" of computation, which normally includes postprocessing of data (such as visualization) on users' local machines. However, these desktops are usually equipped with limited disk bandwidth and memory. Additionally, the persistent storage requirements are greater than what a typical desktop can provide.

A good example illustrating this problem is the setting in a typical national research laboratory, where multiple departments specializing in different disciplines (physics, microbiology, etc.) may exist in close geographic proximity. Each department has a number of workstations connected together in a high-speed LAN, and a multi-gigabit backbone connects these LANs together. Scientists' raw data are usually stored in local or remote high-end mass storage systems, such as HPSS (High Performance Storage System [18]), to which every department has access. However, a scientist is often faced with the task of downloading this data to his/her local machine and performing visualization or some similar interactive tasks. Moreover, it has been shown that researchers working in the same field

make accesses to largely overlapping sets of data [21]. While most desktops are computationally equipped to handle this task, storage requirements are often not sufficient. As a result, a scientist may have to repeatedly download small portions of the data set from the high-end storage repository and run the visualization in multiple inconvenient steps. The high-end mass storage system also generally has a queue of pending requests from other scientists, which adds to the latency of data retrieval. Moreover, data at this end is stored on tape, with the exception of only the most recently accessed data sets which are cached on hard disks. This adversely impacts the data retrieval rate as percieved by the end-user.

The Freeloader project emerged in the hope of addressing the glaring inequities in such settings. While storage space is insufficient for application requirements, most workstations use only a small percentage of their total available disk space [1, 10]. Moreover, the percentage of unused space increases as the capacities of disks on workstations increase. Even when scientists do manage to fit bulk scientific data onto their desktop workstations, they still prefer to delete this data after processing it, inspite of the possibility that it will be used again. Additionally, the disk itself is idle (at a low disk bandwidth utilization) for a large percentage of the time. In a LAN environment, a large number of such user desktops exist, connected together by a high-bandwidth interconnect. In addition, most workstations are online for a majority of the time [6]. As a result, these desktop machines represent good collective storage space. More importantly, these desktops provide very impressive collective disk I/O bandwidth. Freeloader aims at aggregating these precious resources by developing software to manage these distributed idle disk spaces as a single cache and scratch space. We call this process of space aggregation *storage scavenging*, in much the same way that Condor harnesses idle CPU cycles [24].

1.2 Project overview

The FreeLoader project is an effort to aggregate space and I/O bandwidth contributions from commodity desktop workstations in a LAN environment, and to provide a shared cache/scratch space for large, immutable scientific data sets. We designed an architecture and framework that facilitates transient access to such data sets. The aggregated storage space works together with and complements existing and future high-end storage systems, by providing greater throughput and availability as a cache, diverting a significant portion of the burden of servicing data requests to itself. Consider a group of biologists

that routinely use gene sequence databases from a public web repository such as NCBI (National Center for Biotechnology Information) [19]. Extracting meaningful information from these databases, with tools such as pairwise or multiple sequence searches, involves repeated scans over them. Each of these databases can easily be gigabytes in size. Caching them either partially or entirely at a group of local workstations would significantly speed up this process, while reducing the burden on remote storage systems (such as archival systems and web servers) and the wide-area network connection.

With Freeloader, workstation users voluntarily donate storage space. These workstations are called *benefactors*, who run the *benefactor daemon* in the background. Storage is aggregated from multiple benefactors to form the *Freeloader storage cloud*. Data is stored on the benefactors in blocks of data called *morsels*, which serve as the basic units of donated storage. Meanwhile, the *management component*, which runs on one or more dedicated nodes called *manager(s)*, keeps track of benefactors through a registration process. It is also a decision making entity and an information server, which determines data placement on the benefactors and stores/serves such information for data set accesses. The Freeloader *client* is a set of user interfaces between an application accessing Freeloader data and the Freeloader cloud.

Data is either locally generated or staged from one or more remote data sources. Since write-once-read-many accesses are the norm in scientific computing, we design the Freeloader cloud as an intermediate cache. This is especially useful when the remote source is an ftp server or a web repository, which has limited request processing capacity and is easily overwhelmed by user requests, or is connected by a slow network pathway to the user. In addition to space aggregation, Freeloader provides high data retrieval rates by aggregating I/O bandwidths of desktop workstations. By enforcing cache content replacement, Freeloader automatically favors "hot" data sets, saving users' expensive and redundant remote data accesses and data migration operations. Instead, these data sets can be accessed from the Freeloader cloud, potentially at an access rate higher than one single workstation's local disk bandwidth.

1.3 My contributions

I worked with a team in building the Freeloader framework. The underlying communication package, the benefactor daemon, and the communication module for the client

were written by Jonathan Strickland, another student working on the same project. My part of the project has involved the design and development of the manager component and the client interfaces. Listed below are my contributions:

- Manager design
    1. I designed and implemented the soft-state benefactor registration and status update function of the manager.
    2. I implemented the metadata management functions of the manager with a focus on reducing the metadata maintenance overhead.
    3. I implemented a data striping algorithm that exploits parallelism in data access and optimizes space utilization on the Freeloader cloud.
    4. I implemented a cache replacement algorithm based on existing work, to manage the Freeloader cloud as a cache space.
- I designed and implemented the Freeloader client interface APIs with a focus on improving access performance by parallelizing data transfers and enhancing write efficiency with the use of client-side buffering.
- I was also involved in setting up the Freeloader testbed, running experiments through scripts, and collecting results.

1.4 Roadmap of thesis

My thesis is organized as follows. Chapter 2 presents the Freeloader architecture and its major components. We present the design of the manager in Chapter 3. Chapter 4 discusses the design of the client interface. Results of Freeloader performance tests are presented in Chapter 5. In Chapter 6, we discuss other projects and results related to Freeloader. Chapter 7 concludes the thesis and gives a brief discussion of future work.

Chapter 2

Freeloader architecture

2.1 Overview

In this section, we describe the overall architecture of the FreeLoader storage scavenging system by describing its major components and their interrelationships. Section 1.2 gave an example setting that Freeloader is designed to work in. Here, we illustrate the use of Freeloader with a use-case scenario.

Consider a team of biologists within Oak Ridge National Laboratory (ORNL), who want to examine microarray data from a public gene sequence database such as the NCBI's (National Center for Biotechnology Information) GenBank [20]. For example, they may run data mining algorithms on this database to discover irregularities or dissimilarities in the gene expressions, say between normal cells and cancerous cells. The data requirements for this type of application would be in the order of tens to hundreds of gigabytes. The team of researchers needs quick and repeated accesses to this database. The amount of data exceeds the capacity of any individual workstation, but collectively the biology research group has hundreds of gigabytes to terabytes of unused storage on desktop workstations.

Without the Freeloader cache, the situation facing the team of biologists is as follows:

- These researchers have to revise their applications to directly stream-process data from the database hosted at the NCBI web site, at a rate of about 2 MB/s.
- They might resort to replicating this database on their local mass storage system - in this case the HPSS (High Performance Storage System) at ORNL. This gives them an

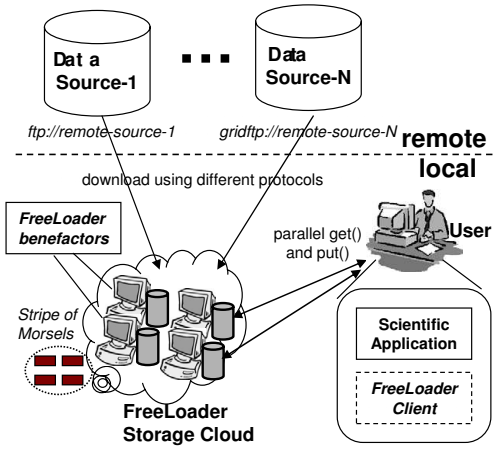

Figure 2.1: Envisioned Freeloader Environment

access rate of about 10 MB/s. Their repeated and frequent accesses to the database places a heavy load on HPSS, which is shared by all ORNL researchers and many external users.

- These researchers can purchase a RAID system or a Storage Area Network solution (SAN) to store such databases. Although disks are inexpensive, disk arrays and SAN products are not. Moreover, shared file systems are often quickly filled up, while the quota system incurs space fragmentation and underutilization.

With Freeloader, these researchers can aggregate their *existing* storage resources for scientific data processing. Before we describe how Freeloader works, we define the important terms that will be used in this context. A *data set* is a sequence of bytes, most commonly a file containing scientific data. Every data set stored on Freeloader is assigned a unique *data set Id*. A *morsel* is the unit of data storage and retrieval on Freeloader. Each morsel of data is uniquely identified by the location independent pair of (*data set Id, block number*). A *block* is a morsel-sized unit of data which refers to the contents of the original data set. We use the terms *block* interchangibly with *morsel*, with the slight distinction that the morsel refers to data blocks which are stored on the Freeloader cloud. The *block number* is the offset of the data block in the data set, in terms of morsel sized blocks. Although we use fixed size morsels in our performance impact experiments, our implementation allows the manager to choose a different morsel size for each data set.

Freeloader alleviates the aforementioned problems of the researchers in the follow-

ing way. When one of the client machines (used by one of the researchers) downloads the public gene database, the client interface also contacts the manager indicating that this data set is to be staged on the Freeloader cloud. Figure 2.1 shows how the Freeloader fits in as an intermediate cache space. The manager responds by creating a list of morsel identifiers for the data set, and in addition decides how morsels of the data set are distributed among benefactors. The manager returns a list of key-value pairs, where each key is the morsel identifier and the value is the network address of the benefactor on which the morsel is to be stored. This mapping between every morsel identifier and its location is maintained by the manager as metadata.

Parallel writes are initiated by the client which ensures that this data is striped onto benefactors. Subsequent requests for access by any one of the team of researchers is sent by their respective client to the manager, which responds with the mapping of the morsels to benefactors. The client uses this mapping to retrieve data in parallel from the benefactors. We optimize Freeloader for write-once-read-many data, which is the norm in scientific data processing [29].

FreeLoader aggregates donated storage into a single scavenged storage system. Storage space is donated by workstation users on a voluntary basis, by running the benefactor daemon as a background process. The basic architecture consists of three main components - the manager, the benefactor, and the client. Figure 2.2 illustrates the relationships between the components. The remainder of this chapter will focus on the functionalities of each of the Freeloader components.

2.2 Manager

The primary function of the manager is to maintain metadata regarding benefactors and data sets. Freeloader's registration process requires each benefactor to periodically contact the manager with statistics such as total storage space donated, used and unused space, its liveness and uptime. If the benefactor already hosts data, the manager stores metadata describing which data sets and their corresponding morsels are currently stored on the benefactor. The manager also maintains metadata specific to data sets. This includes general information such as data set size and location of every morsel of the data set.

Data placement is another major functionality of the manager. This involves

deciding how and where to place data. Before a data set is initially stored on to the cloud by the client, the manager is contacted. The manager chooses a subset of benefactors on which to distribute data. This distribution is also called *data striping*. The manager partitions each data set into equal-sized stripes, and assigns these stripes to a subset of benefactors in a round-robin manner. Each of these stripes consists of one or more morsels, which form the basic units of striping in Freeloader. Striping serves the purpose that subsequent data retrieval by the client can be done in parallel from multiple benefactors. It also serves to optimize space utilization on the Freeloader cloud.

Integral to striping are two basic parameters - *stripe size* and *stripe width*. The stripe size indicates the breadth or the size of each stripe allocated to a benefactor. It is also an indicator of the amount of data retrieved sequentially at a time from a benefactor during retrieval. The stripe width denotes the number of benefactors chosen for the striping of a data set. The choice of these two parameters will be discussed in Chapter 3 and the performance impact of these parameters will be discussed in Chapter 5. Note that the manager is not involved in any morsel data transfer. It only acts as a decision making entity assigning morsels to benefactor nodes as part of the striping process. To access or store the morsels in a data set, a client downloads this mapping of morsels to benefactor nodes from the manager. It then performs a series of *Get()* or *Put()* requests of morsels in parallel.

Striping also achieves the purpose of minimizing the performance impact on benefactors. Even though disk bandwidth is typically considered to be the bottleneck in data accesses, with multiple benefactors serving data concurrently, we can afford to have each benefactor reading at a rate *lower* than its maximum disk bandwidth. Therefore, striping provides a method of saturating the network link on the client side, while at the same time placing an acceptable load on individual benefactors. Striping also achieves some of the future goals of Freeloader, namely high data availability and handling heterogeneity among benefactors. Having variable stripe sizes for different benefactors in proportion to their bandwidth is a promising method of achieving load balancing. Striping achieves the granularity needed for load balancing, replication and relocation of morsels in case the load on a benefactor is excessive. Note that all data placement decisions made by the manager are maintained as part of its metadata.

In addition to deciding data placement and managing metadata, the manager also manages the entire Freeloader space as a cache with an eviction policy. Eviction is mainly

carried out at the level of a data set. The cache replacement algorithm is invoked when there is a space limitation, and a new data set is to be staged on the cloud. At this point, a victim data set is chosen and morsels are evicted in a bottom-up manner from that data set. This is due to the assumption that sequential accesses are the norm and that data sets are accessed in whole rather than partially. We user the LRU-K policy, which is an improvement on the simple LRU in that takes into account a window of past accesses to a data set to estimate the time of the next access. The concepts of replication and relocation are closely related to that of cache replacement. Relocation is based on moving the "hottest" morsels of the "most important" data sets when a benefactor either shuts down or undergoes a space shrinkage. Replication deals with using access history of data sets to create additional replicas of frequently used morsels. Both these features are, however, not currently implemented.

The design goals of the manager are as follows:

- The management scheme should be scalable to manage several hundreds of workstations.
- FreeLoader should be robust and reliable even though the benefactor nodes are not.
- With striping, the data transfer protocol should saturate a client's network connection, providing maximum client-side throughput. This is achieved by a combination of client and manager functionality.
- The benefactor design should accommodate low-end and heterogeneous hardware, while keeping the overhead low so as not to adversely affect user comfort on donated nodes.

More details regarding the design of the manager will be discussed in Chapter 3.

2.3 Benefactor

The benefactor component manages the workstation's donated local storage space. The primary function of this component is servicing morsel *Get()* (retrieve) and *Put()* (store) requests. Because FreeLoader serves write-once-read-many data, we expect read requests to dominate data traffic.

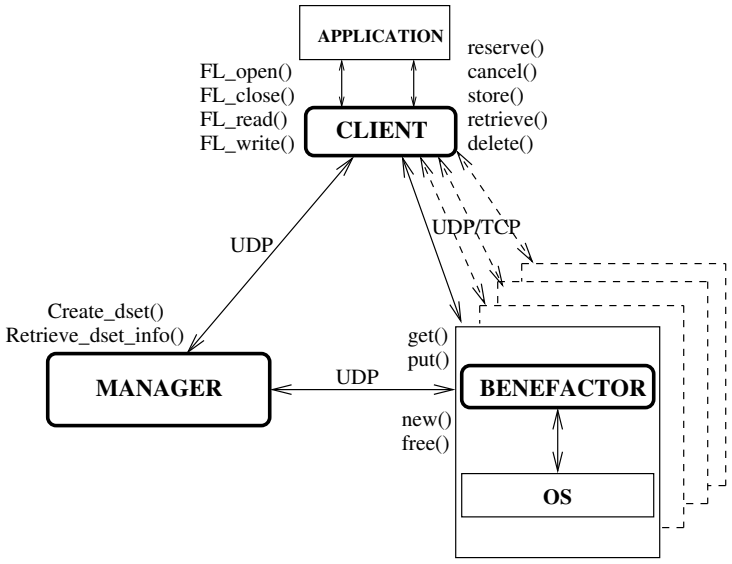

Figure 2.2: Freeloader Components

A benefactor node is an ordinary user machine that is willing to donate its idle disk space and has installed the Freeloader benefactor daemon process, which communicates with the manager and services get/put morsel requests. Freeloader uses *soft-state* registration, where the benefactor daemon regularly sends heartbeat or "I'm alive" messages to the manager to renew its registration in the Freeloader cloud. Aside from servicing requests, the impact of the benefactor daemon on the performance of the host workstation's native workload is negligible.

Since a benefactor node is donated, the resources it offers to the cloud are limited. Moreover, it cannot be considered reliable or robust. Therefore, Freeloader should be designed wtih flexible error handling and recovery capabilities. We choose to use UDP for control messages and metadata transfer for its flexibility, while TCP is primarily used for data transfer between the client and a benefactor.

Another major task of the benefactor is to keep impact on its native workload to a minimum. The daemon must run in the background and co-exist with the native workload. The daemon process monitors the load that FreeLoader is placing on the benefactor and, if necessary, throttles the responses to keep the user discomfort low. As mentioned earlier, striping allows multiple benefactors' data serving capacities to be combined in parallel. This ensures that even when each benefactor is serving data at a rate significantly lower

than its I/O bandwidth, the incoming network bandwidth on the client side can still be saturated. For more details on the benefactor's design and benefactor performance impact control mechanisms, please refer to our research papers [35, 34].

2.4 Client

End-user applications on a client node use the services of Freeloader through Freeloader interfaces. Applications link with the client interface library in order to store to or retrieve data from the cloud. The basic APIs are:

- *Reserve()*: This allows a client to reserve a certain amount of space on the Freeloader cloud.
- *Store()*: When the client decides to store a data set, it contacts the manager with its details. The manager invokes its data striping policy and returns a list of morsels and their assigned benefactors.
- *Put()*: Once the client knows where to place each data set morsel, it may write to each of the benefactors in parallel by calling *Put()*.
- *Retrieve()*: When the client intends to retrieve a dataset from the cloud, it contacts the manager using *Retrieve()*. The manager returns a list of morsel identifiers and the corresponding benefactors on which they are stored.
- *Get()*: Once the client retrieves morsel locations (benefactor node IDs) from the manager, it reads these morsels by initiating *Get()* requests in parallel to each of the benefactors.

Details of the client interface implementation will be discussed in Chapter 4.

Chapter 3

Manager design

3.1 Design Overview

The functionality of the manager can be summarized as that of a lightweight decision making entity and an information server. In our current implementation and experiments, we choose to use only one manager node. However, our design can be easily extended to multiple managers. In particular, our assumption of data being write-once-read-many simplifies issues related to metadata consistency among managers, if more than one exists. The design of the manager is discussed based on the following functionalities:

- Benefactor registration and status tracking
- Metadata maintenance
- Data placement
- Cache management

Prior to discussing each of these topics in detail, we give a brief description of the skeleton of the manager design. Figure 3.1 illustrates this architecture. The manager is built on top of an inter-node communication library which was designed by Jonathan Strickland. This library uses UDP for control messages and TCP for morsel data transfers. Since the manager is mainly concerned with receiving small status updates from the benefactor and requests from the client, it uses UDP as its mode of communication. Communication through UDP involves using generic *Send()* and *Receive()* functions for communicating

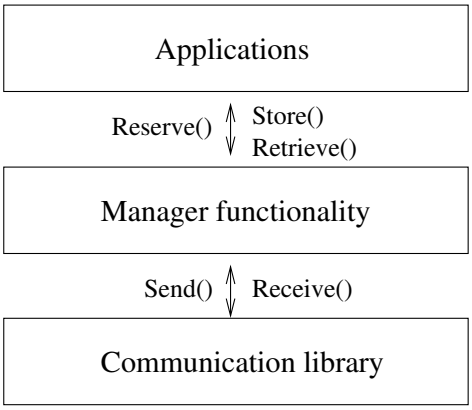

Figure 3.1: Manager architecture

with other nodes. The message Id is passed as a parameter and defines the type of message. Additional parameters may also be specified.

The manager receives requests from the client, services those requests, and responds to the client. It has a well-known address which clients and benefactors have access to. The control flow of the *main()* function in the manager consists primarily of two steps. First, a function handler is registered with the communication library and is associated with a message Id. When an incoming request bears that message Id, the corresponding function handler is invoked. After the message is appropriately handled, the manager returns to sleep(), waiting for another message. The underlying implementation uses the SIGIO interrupt to wake up the manager process and the message Id is used to determine which function handler is invoked. As a result, if multiple messages arrive at the same time, they are placed in a queue and executed in the order of arrival.

The message Ids are integer values which are agreed upon by all communicating parties. An example is the use of messages RetrieveMorselMapping and RetrieveAck. The client sends a message to the manager with the first message Id and data set name as parameter, asking for the mapping of morsels to benefactors. The manager responds with the message RetrieveAck which serves as an acknowledgement, with the mapping of morsels requested.

3.2 Benefactor registration and status tracking

The manager is the only entity in Freeloader which has information regarding all the benefactors. It handles the "membership" of benefactors through a soft-state registration process. Using soft-state registration implies that each benefactor is not "alive" or "valid" unless it keeps itself registered with the manager periodically. The benefactor is responsible for keeping the manager updated regarding its storage capacity, the morsels it hosts and performance history such as the most recent outgoing data delivery rate it could achieve. It is also responsible for notifying the manager in case the amount of space donated is reduced or the benefactor is shutting down. The benefactor sends two types of status messages to the manager:

- *Start-up messages*: Startup messages are sent to the manager to indicate that the benefactor is up and running. Another use of startup messages is to indicate that a benefactor is restarting after a failure or crash. This type of message also carries with it details of data sets and corresponding blocks currently held by the benefactor.

- *Alive messages*: Alive messages are sent to the manager periodically to maintain the benefactor's validity as part of the Freeloader cloud. These messages may contain additional information such as morsels accessed during the last time interval, which could feed into the cache replacement policy.

The mechanism of soft-state registration is used in the manager, which maintains a list of benefactors entries and their corresponding status. Each benefactor periodically (every 20 seconds) sends a status message to the manager. The manager on receiving this status message, sets the status of the benefactor to "alive" in its metadata. The manager also periodically (once every minute in our implementation) checks the status of all the benefactors. It does this by registering a status checking function with the schedular and setting the status check frequency as 1 minute. For all benefactors that are marked as "alive" when the status check is performed, the manager clears their status, and these benefactors are considered to be a valid part of the cloud for the duration of the next minute. For all benefactors marked "invalid" during the status check, the manager sends a status request message forcing a response from the benefactor. Until an update is received from that benefactor, it is not considered a valid part of the cloud.

This is an effective way of implementing soft-state registration, since benefactors can asynchronously send status messages to the manager. Choosing an update frequency for benefactors which is greater than the frequency of status checks performed by the manager ensures that a live benefactor remains a valid part of the cloud. It gives every benefactor the chance of registering with the manager, before it is considered invalid when the status check is performed. The frequency values of 20 seconds and 1 minute remain fixed while Freeloader is in operation. We have determined that with small status messages and a low update frequency, scalability is not a problem even with hundreds to thousands of benefactors.

In the current version of our prototype, the purpose of status messages for data set storage is to determine the list of benefactors on which striping can take place. If a request for a data store arrives from a client, the manager only makes striping decisions based on the set of benefactors that are considered valid at that point in time. For data retrieval, the purpose of the status messages is to determine if the data set is stored on benefactors which are currently invalid. A valid benefactor may still crash during a transfer. Handling of such errors and subsequent recovery are not yet implemented. For future versions of the Freeloader prototype, status updates can be used by managers to determine replication policies for frequently accessed morsels or relocation policies in the event of a benefactor crash, to handle benefactor withdrawal from the cloud or the shrinkage of donated space. Performance histories obtained from these status messages can also help the manager stripe data in smart ways by considering the capability and reliability of individual benefactors.

3.3 Metadata maintenance

The manager is the information server for the Freeloader cloud. It must maintain metadata regarding benefactors and data sets. Additionally, the manager needs metadata for its own purposes, such as using a history of statistics on data access performance and benefactor availability. Freeloader's metadata management has to take into account several issues. First, metadata must remain persistent at the manager node to make it capable of failure recovery and restoring the Freeloader system status. Second, the size of metadata should be reasonable, and metadata look-up should be efficient. As the single metadata server in the system, the manager should be able to handle dozens to hundreds of benefactors and clients with typical scientific workloads, without becoming a bottleneck in overall per-

formance. The manager must also support highly generic *store()*, *retrieve-by-key()*, *update()* and *delete()* functions. Portability to various UNIX compatible platforms and Windows is also important.

Given these factors the metadata management component of the manager must have the following characteristics. First, it should have basic database functionality - an example of this would be storing benefactor Id and the corresponding available space as a record. The benefactor Id serves as the key in this case. Another example is the mapping from a unique data set block to a specific location/benefactor. In this case the pair of attributes - (dataset Id, block number) represents the key. Second, the retrieval must also be fast. This implies that the data management uses some form of indexing. The data management utility must have the ability to handle complex data types - an example is the 4 octets used in representing a node's IP address. Database functionality should be obtained by just linking with a library. Some of the more mature and complex database systems like MySQL and PostgreSQL introduce too much overhead, due to the fact that they provide for more complex functionality like security and consistency. Moreover, they need to be installed as servers before they can be used.

We use the *dbm* package because it has the above features. Apart from its ease of use, it allows the user to define the structures of records and a key field by which to index the records. It also provides a set of generic APIs which can be used to store records, delete records, traverse the database record-by-record, fetch records by key, and update records. UNIX uses the *dbm* package for a number of its utilities. Command line utilities on Linux commonly use *gdbm*, which is a lightweight indexing database package. We use the *ndbm* package, which is a close variant of *gdbm*. It has been proven to be sufficiently fast in terms of fetching by key. To illustrate this, consider a test case involving look-up of morsel locations for a data set of size 5 gigabytes. Over multiple retrievals of this 5 gigabyte data set, the manager overhead in terms of look-up never exceeded 0.15 seconds. Even with a high aggregate throughput of 45 MB/s from 4 benefactors, this represents less than 0.15% of the total retrieval time. File storage overheads are also minimal. For the data set of size 5 gigabytes and morsel size of 1 MB, the associated metadata is less than 800 kilobytes. This is a reasonable amount of space, considering the fact that the manager is a dedicated machine. Note that this is inclusive of the per-morsel data mapping, which we store for better data placement flexibility. Metadata size will decrease if the morsel size is increased. Each database is stored in the form of two files - a sequence of records and

an index file. This package is used in our implementation for all benefactor-specific and data set-specific metadata. These include mappings from data set Ids to file names, data set striping parameters, mappings from morsels to benefactors, and the benefactor-specific status information.

3.3.1 Morsel metadata

All data is stored on the Freeloader space in morsels. Apart from storing data specific to data sets, the manager also keeps information specific to individual morsels. The morsel specific information stored by the manager includes location of all replicas of the morsel in the form of the benefactor network addresses.

Associated with each morsel is also its *heat*. This represents the frequency of accesses to the morsel, divided over all replicas of the morsel. This forms a good approximation of the demand for a particular morsel. When the number of replicas reduces, the demand increases. Data sets and their corresponding morsels' heats must be recomputed constantly when they are accessed from the Freeloader space. This *heat* information is an important basis for the manager to make decisions regarding eviction, relocation and replication. Relocation of morsels is carried out when either a benefactor quits or reclaims its donated storage, or the amount of space donated by the benefactor is reduced on demand by the workstation user. Relocation involves moving the "most important" morsels from such benefactors to other benefactors in a time-critical manner. Replication involves storing duplicates of morsels on the cloud based on their "hotness" or frequency of use.

A simplistic explanation of the above strategy can be stated as - morsels with the greatest demand are the first candidates for relocation and replication, and the least preferred candidates for eviction. This *heat index* is expected to form the basis of cost model-based strategies in future versions of the manager, where replication and relocation are incorporated. Although relocation and replication are not currently implemented, the data structures we use are designed to accomodate this future functionality.

Morsel metadata is periodically forced to persistent storage. However, the morsel mapping functionality uses metadata stored (cached) in memory to minimize response time as seen by the client. There is a need for choosing data structures which incur minimum overhead in the look-up process. With naive linked list implementations, there are potential scalability problems because the volume of incoming status updates can be very high. Heat

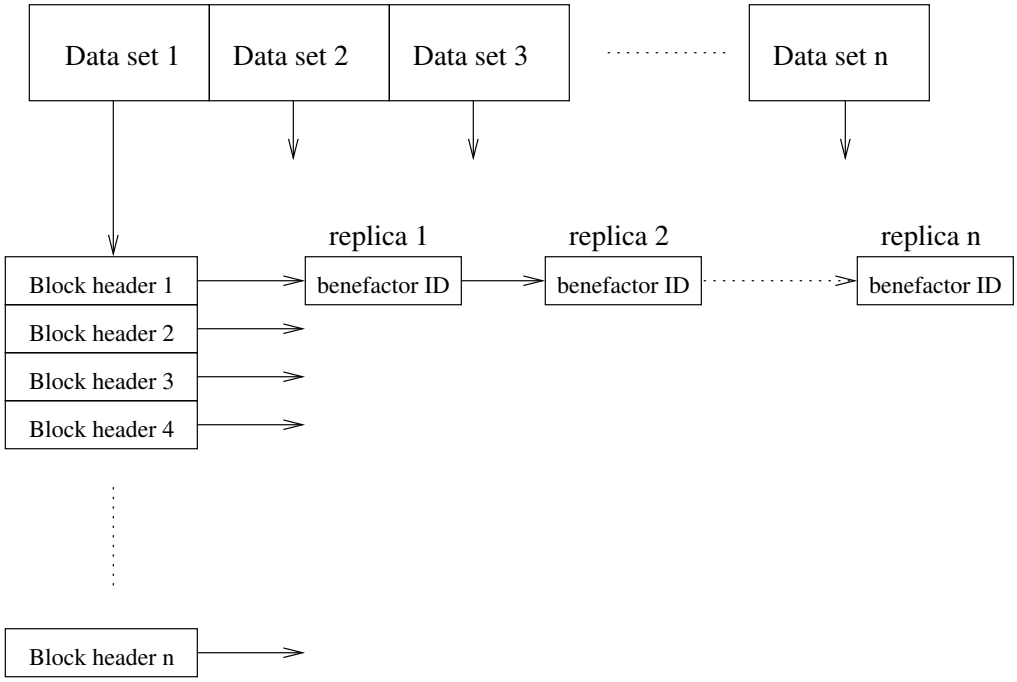

Figure 3.2: Data structures for morsel metadata

indexes need to be recomputed, and they may vary from one morsel to another. This can be a problem when there are many benefactors, possibly in tens or hundreds.

The structures we use are illustrated in Figure 3.2. We use an array of pointers, each element of which is assigned to a data set. This array is dynamically allocated since the number of data sets is not known in advance. Each pointer points to the base address of an array of blocks which belong to that data set. Each element in the block array corresponds to a block number of the original file. This method of dynamically allocating arrays for blocks proves to be efficient since the size of a data set is not known in advance. This also provides look-up performance which is similar to using a hashing scheme. For example - *dataSet[X].blockNumber[Y]* refers to the node which represents block number Y of data set Id X. This node refers to a linked list of replicas of that morsel. Each node of the linked list represents a different replica and stores the benefactor network address where it is stored. The ordering of replicas is based on the time of last access to each replica. Since look-up is inexpensive, updates can be carried out quickly. Deletion or invalidation of a data set is a simple process of marking the data set as invalid by using a flag.

3.4 Data placement

Data placement is crucial to the data access performance of Freeloader. The data placement function comes into play when the manager receives a message from a client for storing data. The client supplies the data set size, and the manager chooses a suitable morsel size, stripe size, stripe width and a subset of benefactors on which to stripe that data set. The client expects in return a list mapping each morsel of the data set to the benefactor on which it is stored, so that it can proceed with storing these morsels on those benefactors.

The manager's role in data placement is to serve two purposes. First, it needs to stripe data in a manner so as to maximize the client perceived retrieval bandwidth. The stripe size, stripe width and morsel size are three parameters which need to be chosen by the manager prior to striping. The stripe size defines how many consecutive morsels are to be stored on one benefactor, and the stripe width defines the number of benefactors across which the data is striped. We use an emperical study to determine the optimal morsel size, by increasing morsel size in powers of 2, starting from 16 KB, and studying the impact on retrieval performance and size of metadata. The size of metadata, in turn directly affects the metadata storage needs and the overhead of the manager in the process of look-up of morsel locations. In our experiments, we use a morsel size of 1 MB as a value that provides enough granularity for striping, and is still large enough to keep metadata storage and look-up overhead minimal. Currently, stripe width values for data sets are chosen as powers of 2. We find that retrieval performance is maximized between a stripe width of 4 and 6 benefactors. Benefactors are assumed to be similar and therefore a uniform stripe size is used across all benefactors. However future versions of the manager will assign a stripe size which is commensurate with the retrieval rate of a benefactor. The impact of varying stripe width and stripe size is shown in Section 5.2.

The second goal of data placement is to optimize the utilization of the storage capacity available on the Freeloader cloud. The manager needs to use a suitable striping algorithm. Informally, the problem of striping can be stated as follows. Given the set of m benefactors with their corresponding storage capacities, and an incoming data set d of size s - choose a subset of the benefactors on which to stripe the data.

We now formally describe the problem of creating an optimal fit of data, given the size of the data set which is to be stored. Consider that the Freeloader cloud is made up of

```
Get updated list of benefactors B[], and corresponding free space F[]
Sort benefactors in descending order of free space
MorselCounter = 1
MorselLocation[i] gives the benefactor Id on which Morsel i of data set is stored
Do
      For i = 1 to StripeWidth
            For j = 1 to StripeSize
                  Assign the morsel numbered X to B[i], where X = MorselCounter
                  MorselLocation[MorselCounter] = B[i]
                  MorselCounter = MorselCounter + 1
                  Update F[i] = F[i] - MorselSize
            End For
      End For
Repeat until all morsels have been assigned to benefactors
Array MorselLocation[] contains the morsel mappings
```

Figure 3.3: Striping algorithm

m benefactors $B = \{b_1, b_2, \ldots, b_m\}$, where benefactor b_i has initial free space size f_i. The input consists of a sequence of n datasets $D = d_1$, d_2, ..., d_n, where d_i is of size s_i and uses a stripe width w_i. The problem is one of striping as long a prefix of D to B as possible. We have shown that a known NP-hard problem, *Minimum Bin Packing*, can be reduced to the off-line version of the above problem.

The manager handles this using a greedy heuristic. Given a list of m benefactors $B = \{b_1, b_2, \ldots, b_m\}$, where benefactor b_i has initial free space size f_i, we sort the benefactors in decreasing order by their available unused space. This represents the most up-to-date "view" of the cloud at any moment of time. Given an incoming data set d of size s and stripewidth w, the top w entries from the sorted benefactor list are chosen for the striping. The striping is also dependent on the sufficient availability of storage in top w benefactors in the list. In the event that the space is insufficient after one round of striping, another round of striping is carried out on the top w benefactors for striping the remainder of the data set. Once the subset of benefactors is chosen, the manager uses a round-robin strategy to assign consecutive stripes of the data set to different benefactors. Each stripe consists of *stripesize* number of morsels. Once this striping is done, the morsel mapping is sent to the client. The rationale behind using the heuristic is as follows. Assigning the new data

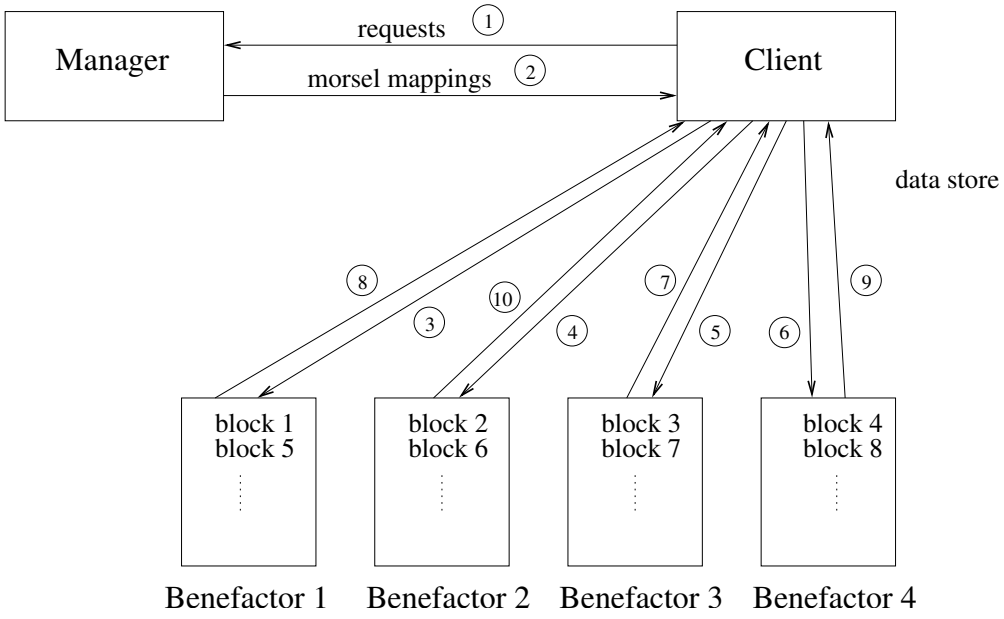

Figure 3.4: Example of striping (stripe width = 4, stripe size = 1 morsel)

set to the top w benefactors in the sorted list reduces the imbalance in storage capacity among benefactors, making the cloud of benefactors more amenable to uniform striping. Pseudo-code of the striping algorithm is presented in Figure 3.3.

Figure 3.4 illustrates an example of how the blocks are distributed when the client uses the mapping to store data. The directed edges are labelled with numbers which are indicative of the order in which the corresponding events occur. The client first contacts the manager with details of the data set which it intends to store, requesting details on how and where to store the data set on the cloud. The manager then chooses the appropriate morsel size, and also chooses the striping parameters - stripe width and stripe size. The manager applies the greedy algorithm described above and assigns morsels to benefactors. Note that this is an allocation process and does not mean that morsels are actually being placed on benefactors. This allocation or mapping is returned to the client which stores morsels on the benefactors using the *Put()* interface. Steps 3, 4, 5 and 6 denote calls of *Put()* by the client on individual benefactors. These steps are overlapped with respect to each other. The acknowledgements may not however, arrive in the same order. After storing block 3, benefactor 3 is the first to reply with an acknowledgement labelled 7. The figure shows that acknowledgements are received by the client from benefactors 3, 1, 4 and 2 in

that order. The client initiates new morsel transfers for benefactors in the order in which it receives acknowledgements from them. This means that the next morsel stored on the cloud is block number 7 to benefactor 3, followed by block number 5 to benefactor 1, and so on. This example shows how more than one benefactor might be storing data in parallel, while the client itself reads another morsel to store onto the cloud. This parallelism is achieved through a combination of striping done by the manager and the data transfer protocol used by the client.

3.5 Cache management

The entire Freeloader space is a finite, limited resource which must be managed effectively by the manager. New requests can be expected from clients at any time. Since the Freeloader cloud acts as a cache of frequently accessed data sets, a cache eviction policy is needed. A large number of policies came into contention initially, ranging from the basic LRU (Least Recently Used) to the Least Frequently Used (LFU) policies. In our implementation, the policy chosen for cache eviction is a variant of the LRU-K replacement policy [28].

The basic principle behind LRU-K replacement is the eviction of the data set which has the greatest backward-K distance. Accesses to data sets can be represented as a reference string of accesses. The backward-K distance for a particular data set is the time from the last access to the K-th most recent reference to the same data set. When a client makes a request for storing a new data set and the space on the Freeloader cloud is insufficient to host the new data set, the cache eviction policy is invoked. The one chosen to be the victim data set is that which has the greatest backward-K distance. The rationale is that backward-K distance acts as a good estimate of inter-arrival (time of next access) time of data set accesses.

Once a victim data set is chosen, morsels are evicted from the bottom up. This means that if data set d has n morsels, the order of eviction of morsels is n, $n-1$, and so on until the space evicted is sufficient to accomodate the new data set. It is possible that eviction of one data set does not free enough information for the new data set. In this case, the next victim data set is chosen based on the same criteria as before - backward-K distance. The reason for evicting blocks in the bottom up manner is that sequential accesses are the norm in scientific data sets. The morsels at the beginning of a data set are often

needed first. This order of eviction allows the data recovery from a remote primary copy to overlap with data accesses within Freeloader. Partial data set recovery, however, is not currently implemented and forms part of our future work (Chapter 7).

The reasons for choosing LRU-K as our eviction policy are as follows. Simple LRU only uses the last access to a data set to estimate the arrival time of the next access to the data set. LRU-K, on the other hand, uses the K most recent references to the data set as an estimate of when the data set will be accessed next. Therefore, it also has a built-in notion of aging, by considering only the last K references to a data set. Unlike LRU-K, LFU is unable to adapt to changing access patterns. LRU-K does not rely on external hints about workload characteristics. It is also well suited to the case where there are repeated, and often periodic accesses to the same data set.

We implement a slightly extended version of the LRU-K algorithm. Backward-K distance is determined by subtracting time stamp values, which are recorded with every data set access. This serves the same purpose as a reference string of accesses. In our implementation of the LRU-K algorithm, we choose the value of K as 3. This is based on observations made by the authors of the paper describing the LRU-K algorithm [28], where they indicate that LRU-2 is no worse than LRU-K (where K is greater than 2), especially for changing access patterns. Since access patterns may remain constant over a period of time, and choosing a greater value of K incurs greater overhead, the value of K is arbitrarily chosen as 3.

The manager also needs to protect data sets from eviction under some circumstances, such as when they are being accessed. This is handled by the client sending explicit messages to the manager to indicate the start and end of a data set store or retrieval. While a data set is being accessed, it is protected by the manager. Therefore, there is no danger of a data set being evicted while it is being retrieved. Data sets which have recently been placed on the cloud also tend to be targeted for eviction, due to their backward-K distance of infinity. Intuitively, such data sets may be considered "cold" since there have been few acccesses (less than K, possibly none) to them. As a result, with the standard eviction algorithm, it is not uncommon for a data set to be evicted shortly after it is stored on the cloud. These data sets need to be protected by the manager since each data set is expected to be used at least once after it is stored in Freeloader space. The manager uses a time-out value, which is twice the average time difference between the store and the first access as observed for all previous data sets. After a data set is placed on the cloud, it is locked

(protected from eviction) for the period of the time-out value, after which it is subject to the same criterion as used by the regular LRU-K policy. It is also possible for more than one data set to have a backward-K distance of infinity. This is because the data set may not have been accessed K times since it was placed on the cloud. In such cases, simple LRU is used to choose the victim from among these data sets.

Chapter 4

Client design

On the client side, a user application invokes Freeloader client interfaces such as *Reserve()*, *Store()*, *Retrieve()*, *Get()* or *Put()*. These functions are implemented as part of the interface library. The interface library also implements a buffering strategy to ensure that the retrieved file is written out in an optimized manner in terms of disk seeks. Such buffering is necessary to ensure that the packets received from multiple benefactors are correctly ordered before the application processes the data. In this chapter we briefly discuss the implementation of the basic APIs provided by the client interface. We also discuss the specific buffering strategy used in the client library implementation. Note that the term *write* is intended to mean either writing to disk or writing to the input stream of the application.

4.1 Implementation of client services

All of the client services are implemented on top of the communication library mentioned in Chapter 3. *Reserve()* is implemented using a simple request response exchange. The client requests a slot on the cloud for a particular data set, and the manager responds with a data set Id if the reservation is successful. The client uses this Id for all future references to the data set. Reservation is granted based on the amount of space requested by the client. In our implementation, we reject a reservation request if the client requests more than one third of the total available Freeloader space. This prevents any one data set from dominating the storage provided by Freeloader. Once a reservation is made, it remains valid until the client explicitly cancels the reservation. We can afford to do this

because space allocation does not begin until the client calls *Store()*.

By calling *Store()*, the client requests an allocation of its data set by sending data set details to the manager. The manager performs the striping and responds with a list which maps morsel Ids to the network addresses of the benefactors on which they are to be stored. The client uses *Retrieve()* to request the mapping of morsels for a data set already stored on Freeloader space. The manager performs a look-up on its metadata and responds with a mapping of morsels. Therefore, *Store()* and *Retrieve()* share some common functionality in that both expect a list of morsel mappings in return.

Once the morsel mappings are obtained, processing is done by the client to divide the morsels into separate streams, one per benefactor. This is done by scanning the ordered list of morsel-to-benefactor entries (ordered by block number) and assigning all morsels located in a benefactor to a separate array. This array represents the stream of data to be stored to or retrieved from that particular benefactor. Once this initial processing is done, the client calls *Put()* for storing data or *Get()* for retrieving data.

- *Put()*: Morsel data corresponding to the first morsel of each stream is sent out onto the network. When an acknowledgement is received by the client from a benefactor indicating a morsel was received, the client sends out the next morsel to the benefactor corresponding to that stream. The implementation allows for concurrent morsel data transfers from the client to each of the benefactors involved, thus making maximum use of the client's network link.

- *Get()*: Requests for the first morsel of each stream are sent out to the corresponding benefactor. Since this process is non-blocking, these requests are serviced by individual benefactors simultaneously. This allows benefactors to overlap their disk I/O and network activity, thus allowing for concurrent morsel data transfers from multiple benefactors. When a particular morsel of information is received by the client, it sends out the next morsel request to the benefactor corresponding to that stream. On receiving a morsel, the client does not immediately write it to disk or the application's input. Instead, it buffers the morsel, waiting for the opportunity to do a consecutive write with one or more received morsels to emulate a sequential stream of data. This is discussed in the next section.

The method used in both *Put()* and *Get()* involves waiting on a response from a benefactor before sending out the next request to that benefactor. This ensures that the

```
CurrentBlockNumber = first block
Grab stripeWidth buffers from buffer pool
Associate buffers with morsel requests
Send initial morsel requests to all benefactors
Loop:
    Wait on any incoming morsel (blocking call)
    Store morsel in associated buffer
    Identify source benefactor of received morsel
    Grab a buffer from buffer pool for next request
    If grabbing was successful
         Associate grabbed buffer with next request
         Send next morsel request to the source
     Else
         If CurrentBlockNumber is not yet received
              Skip CurrentBlockNumber and write first received block to disk
         Else
              Write CurrentBlockNumber to disk
         Grab the buffer just released for next request
         Associate grabbed buffer with next request
         Send next morsel request to the source
         Update CurrentBlockNumber to point to first unwritten block
Repeat loop while there are pending morsels
```

Figure 4.1: Pseudo code for *Get()*

rate at which the client sends or receives information is controlled. This is particularly crucial in *Get()*, where it is important not to swamp the client with too many incoming morsels. The pseudo code for *Get()* in conjunction with the buffering strategy can be seen in Figure 4.1.

4.2 Buffering strategy for data retrieval

A fixed pool of buffers is preallocated to store incoming morsels. Each buffer accomodates one morsel. We use $stripewidth \times (stripesize + 1)$ number of buffers for retrieval, each of which equals the size of a morsel. Therefore we can potentially have *stripesize* morsels from each benefactor in the buffer without having to write to disk or the application. An increase in either the stripe width or the stripe size increases the size of the buffer pool allocated and the memory requirements of the client interface. Prior to sending out a request for any morsel, a buffer from this pool is assigned to the outgoing request.

When data arrives, it is stored in the buffer associated with its request. The method used can be thought of as a double-buffering scheme, where the previous buffer is being processed by the application while the next buffer is being assembled.

When the buffer pool runs out of free buffers, a write must be forced to free up buffer space. This might mean skipping some holes in the data while carrying out the write. When the morsel corresponding to the hole arrives, a seek is done backwards to fill up the hole and write out the morsel. The buffering strategy described here is a best-effort attempt to write out morsels in a consecutive manner. When morsels arrive out of order, they are buffered. When the buffers run out, there might be some out-of-order writing, but only to ensure free space on the buffer pool. With an increase in stripe size, the likelihood of the occurrence of a hole in the buffer increases. This is the reason we use only $stripesize + 1$ buffers for each benefactor to which the client connects to. Any higher value would most likely result in an increase in the number of holes and therefore the number of out-of-order writes of data. Additionally, when the morsel corresponding to the hole arrives, the length of the disk seek is greater, resulting in a decrease of write efficiency.

Chapter 5

Results

In this section, we present the results of using the Freeloader on different data sets. We will first describe the testbed on which these tests are conducted. We then present a comparison of the performance of traditional data retrieval techniques with that of retrieval from the Freeloader cloud. Finally, the impact of variation of stripe width and stripe size on the retrieval performance will be discussed.

5.1 Testbed

The Freeloader cloud tests were carried out at Oak Ridge National Laboratory (ORNL) and North Carolina State University (NCSU). The cloud consists of a dozen machines at ORNL, each of which is a Pentium III, dual processor machine running Linux 2.4.20-8. Each of these nodes has a 100 Mb/sec Ethernet card connected to a Gigabit hub. This means that the network outflow rate of each benefactor is limited to 100 Mb/sec, which is less than its disk bandwidth, which averages between 20 and 30 MB/sec. Each benefactor makes a contribution of between 7 and 30 GBs of free disk space, which adds up to about 120 GB overall. Figure 5.1 shows the testbed used to carry out the experiments.

The following comparison tests are also carried out:

- Retrieval of data sets from HPSS (High Performance Storage System) which has petabytes of storage capacity in the form of tapes.
- Retrieval of data sets from one of ORNL's IBM SPs, served by the GPFS parallel file system. Retrieval is carried out via GridFTP.

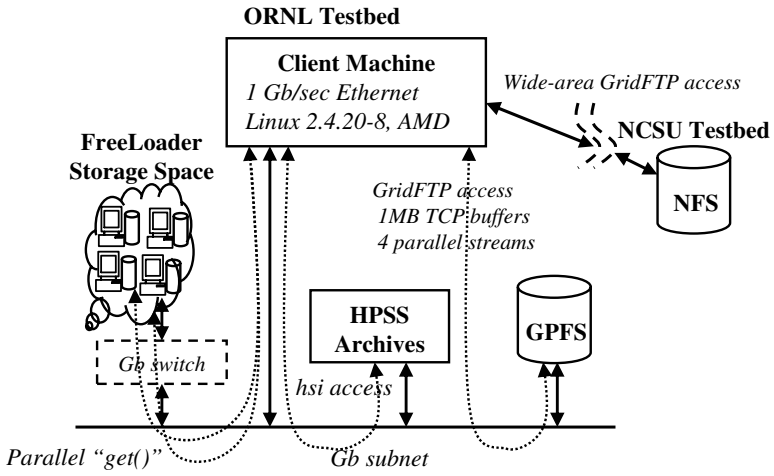

Figure 5.1: Freeloader testbed

- GridFTP access to data sets stored on the NFS storage at NCSU's High Performance Computing Center blade cluster.

For tests involving GridFTP access, tuned TCP buffer settings and parallel streams were used in order to have a more competitive setting for Freeloader. With HPSS, we recorded data for "cold" and "hot" retrieval separately. This was done in order to account for the impact of the disk cache used in HPSS for frequently accessed data. Data was also retrieved from the NCBI website `http://www.ncbi.nlm.nih.gov` using *wget* to download the data. This represents the typical method of retrieval from a remote server.

Table 5.1 shows the entire suite of tests that were carried out. File sizes were varied from 256 MB to 2 GB. Tests were conducted over a three-day period with each file transfer being repeated between 30 and 50 times. A common client machine was chosen for all these tests. The client has a similar configuration to the benefactor workstations, except that it has a Gigabit network card. The benefactor machines, the client machine, HPSS and the IBM SP are all connected by a gigabit subnet, which in turn is connected externally by an OC-12 link.

5.2 Freeloader performance

In this section we compare Freeloader's performance with other retrieval methods and the impact of varying stripe size and stripe width on the retrieval performance of

FreeLoader	12 Benefactors, 1 Manager and 1 client
(Data set size)	256MB, 512MB, 1GB, 2GB
(Morsel size)	1MB
(Stripe size)	1MB, 2MB, 4MB, 8MB, 16MB, 32MB
(Stripe Width)	1, 2, 4, 6
GPFS	GridFTP, GSI authent., 1MB TCP buf., 4 streams
NFS	GridFTP, GSI authent., 1MB TCP buf., 4 streams
HPSS	hsi with DCE authentication
(Hot)	Data maintained in disk caches of HPSS
(Cold)	Fetch from tape: data purged from caches before fetch
NCBI	wget from `http://www.ncbi.nlm.nih.gov`

Table 5.1: Throughput test setup

Freeloader.

Figure 5.2 clearly shows that Freeloader outperforms GridFTP by a factor of 2 and HPSS by a factor of 4. Retrieval using *wget* understandably shows the poorest performance. GridFTP access to a remote NFS server is a little better due to the tuned buffer settings and the parallel streams. HPSS cached data is retrieved at a faster rate than data retrieved from archival tape, which can also be expected. The most significant result is that Freeloader outperforms GridFTP access to the local GPFS file system. This justifies the use of Freeloader as a high-performance cache space.

As mentioned earlier, experiments are conducted with data sets of varying file sizes. For each file size, further tests are conducted by varying stripe size and stripe width. In our tests, the stripe width chosen is either 1, 2, 4 or 6 benefactors. The morsel size chosen and used consistently through the tests is 1 MB. The stripe size is an integer multiple of the morsel size - 1, 2, 4, 8, 16 or 32 MB. The morsel size was fixed at 1 MB after considering the trade-off between increased flexibility for smaller morsel sizes and reduced book-keeping with larger morsel sizes.

Figures 5.3 and 5.4 show the impact of varying the stripe width on retrieval performance with fixed stripe sizes of 1 morsel and 8 morsels respectively. The following observations are noted from these graphs:

- Retrieval performance speedup is linear while varying from 1 benefactor to 2 benefactors, and then to 4 benefactors. There is a speedup when 6 benefactors are used for striping, but the improvement is not a factor of 2.

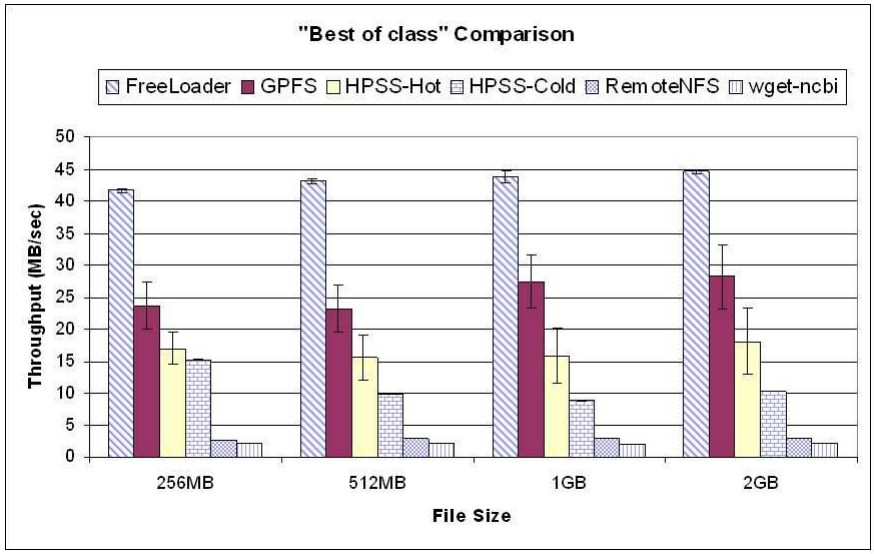

Figure 5.2: Best of class throughput comparison, with 95% confidence ranges

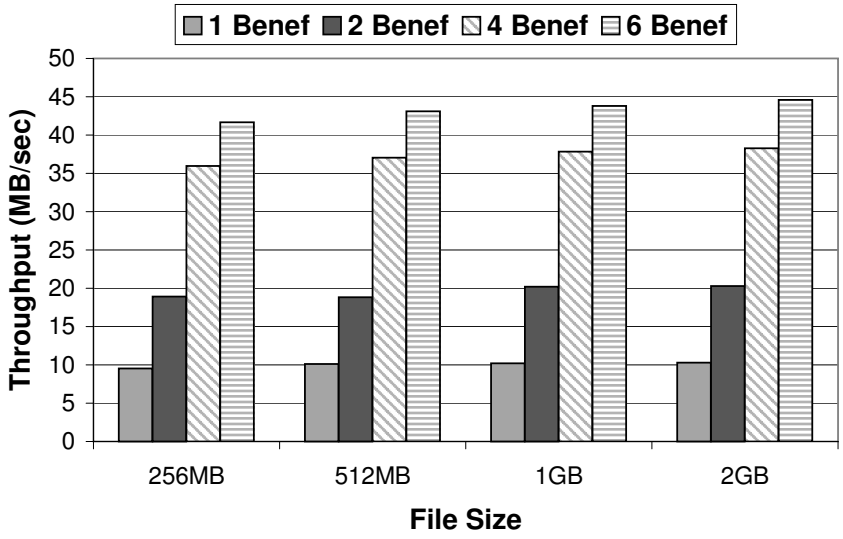

Figure 5.3: Impact of stripe width variation (stripe size = 1 MB)

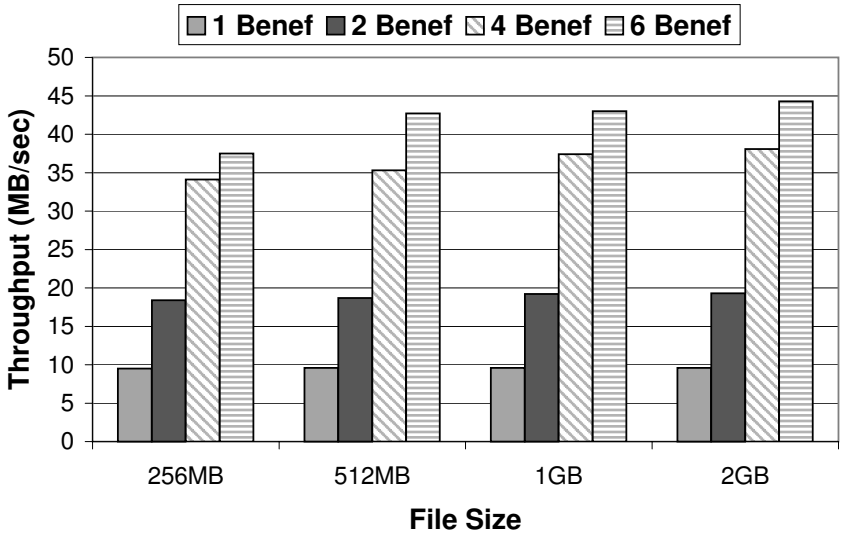

Figure 5.4: Impact of stripe width variation (stripe size = 8 MB)

- The speedup is consistent over different file sizes.
- There is a slight improvement in performance as there is an increase in file size, for any particular stripe width. For example, for a stripe width of 4 there is a slight improvement in overall throughput when file sizes vary from 256 MB, to 512 MB, through 1 GB and 2 GB. This is seen in both graphs.

The cause for the linear speedup is due to the parallelism in data retrieval. The client initiates parallel transfers from benefactors containing the striped data. Different benefactors then begin to read morsels from their own donated space and send packets over the network in parallel. This is one of the main reasons that Freeloader outperforms some of the other retrieval methods. The next section will discuss this in further detail.

The client link is saturated between 4 and 6 benefactors, with the individual benefactors delivering data at their full capacity (10 MB/s). However, based on the load on the benefactors and their capacities, we can afford to increase the stripe width and reduce the rate at which data is delivered from each benefactor. This would serve to reduce the impact on individual benefactors, while keeping the retrieval rate high. The third observation listed above can be explained as follows. As the file grows larger, the initial overhead of retrieving the morsel mapping from the manager is amortized over a larger file size. As a result, the total throughput increases. The fact that the increase is very small

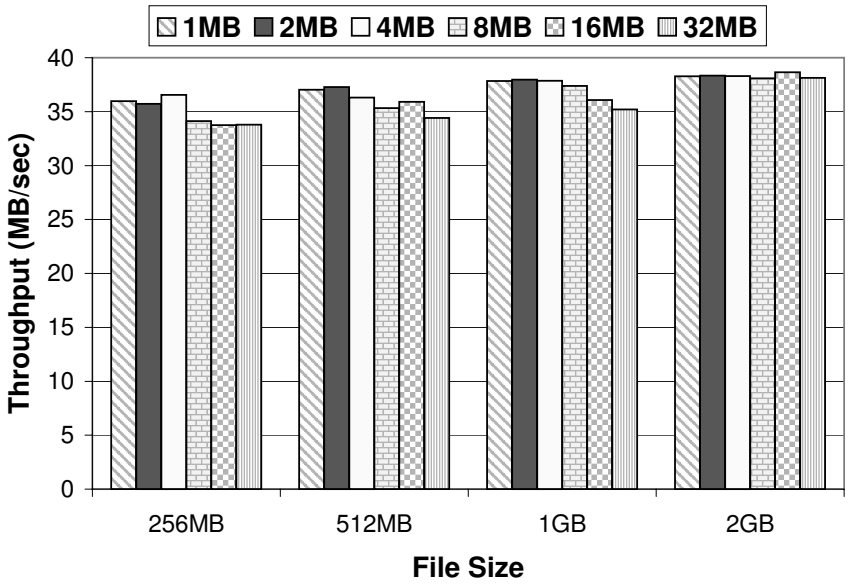

Figure 5.5: Impact of stripe size variation (stripe width = 4 benefactors)

and hardly perceptible, is a testament to the fact that the manager overhead in the overall retrieval is minimal.

Figure 5.5 shows the impact of variation of stripe size on the overall retrieval. As seen in the figure, an increase in stripe size does not seem to affect the overall retrieval rate. This is a favorable result for 2 reasons. Firstly, it implies that the parallelism in data retrieval is very effective. Keeping the stripe size small means that the client switches between packets of different benefactors more frequently. One would normally expect that this would adversely affect retrieval performance as opposed to the case where the stripe size is large and each benefactor reads and transfers a larger amount of sequential data. However, the graph does not show any clear correlation between an increase in stripe size and retrieval rate. Secondly, this also means that the memory requirements on the client side are reduced. The buffer size allocated on the client is set to $stripewidth \times (stripesize + 1)$ morsels. A smaller granularity of striping also means that future tasks such as replication and relocation become easier and more flexible. We currently stripe data over a fixed set of benefactors (*stripewidth* number of benefactors). With smaller stripe sizes, striping over variable sets of benefactors becomes possible, while still maintaining the parallelism in retrieval.

Chapter 6

Related work

A number of projects closely resemble the Freeloader project. What separates Freeloader from the other projects is its unique combination of features and that it is not a file system. Freeloader is an effort to be aggressive in disk bandwidth borrowing from workstations, while still attempting to keep impact on native workstation tasks to a minimum. Results from [16] clearly show that for normal user tasks, borrowing disk bandwidth can be aggressive without creating any perceivable impact. This is also true for borrowing physical memory space and CPU cycles, though to a far lesser extent.

The Google File System [13] is a scalable, distributed file system which is developed exclusively for Google's data-intensive applications. The file system is built on top of a large number of commodity machines, where failures are the norm. Data sets are extremely large in size and are mostly read, rather than written to. High sustained bandwidth is more important to their applications than low latency. The Google File System also shares some other commonalities with Freeloader. Chunk servers are used which are similar to the benefactors in our setup. Replication is used to increase the availability. A single master is used to maintain metadata, and all data transfers take place directly between the clients and the chunk servers. TCP connections are used for data transfer and are kept open over a period of time. Freeloader differs in that it is not a file system, and therefore does not deal with more complex file system issues like consistency, name spaces or atomicity of operations.

Farsite [1] is another project that shares some of Freeloader's goals. It is a scalable, serverless distributed file system that functions logically as a centralized file server. Due to the fact that metadata is distributed among many machines, consistency is an important

issue. The integrity of directory and file data is maintained through a Byzantine-fault-tolerant protocol. The Byzantine Generals Problem [23, 5] is one of the many problems that Freeloader faces if multiple managers are used. Security is also considered a major issue in the design of Farsite. In the Freeloader project, we assume that since data is staged within a relatively secure domain, security is a secondary issue. Farsite is also not designed for high-performance I/O needed by scientific applications.

The Andrew File System (AFS) project [17] is a location-transparent distributed file system that has been under development at Carnegie Mellon University since 1983 and is one of the first successful full-featured distributed file systems. AFS is composed of cells, with each cell representing an independently administered portion of file space. Cells connect to form one enormous UNIX file system under the root /afs directory. An AFS cell is a collection of servers grouped together administratively and presenting a single, cohesive filesystem. The main strengths of AFS are its client-side caching facility, its scalability and location independence.

Network File System (NFS) [31] was designed to provide transparent, remote access to shared filesystems across, with a special emphasis on portability and using the platform neutral XDR (External Data Representation). NFS is built on top of Sun RPC and is a stateless protocol. The primary functions of NFS are to export or mount directories to other machines, either on or off a local network. These directories can then be accessed as though they were local. NFS uses a client/server architecture and consists of a client program, a server program, and a protocol used to communicate between the two. The server program makes filesystems available for access by other machines via a process called exporting. NFS clients access shared file systems by mounting them from an NFS server machine. When a file system is mounted, it is integrated into the directory tree and accessed as a different mount point. One characteristic feature of NFS is the Virtual File System (VFS) and the use of Vnodes which define operations on files.

Coda [32] is a file system for a large-scale distributed computing environment composed of Unix workstations. It provides for high availability and performance by using server replication which involves storing copies of a file on multiple servers. It also uses disconnected operation, a mode of execution in which a caching site temporarily takes on the role of a replication site. Clients cache most of the data needed to locate files. When a connection to a server breaks down, the file system continues to provide the same service, while at the same time logging the changes. On reconnection, it brings the server up to date.

Security is achieved through Kerberos-like authentication and access control lists. Coda is a descendant of AFS-2, but is more robust than AFS when it faces network problems and server failures. The primary thrust of Coda is to support mobile computing. LOCUS [30] is a Unix compatible distributed operating system which provides a single image of a file system inspite of files being distributed among multiple machines.

Kosha [3] is a P2P enhancement to regular NFS. It aggregates free storage from cluster nodes and desktop machines to form a low-overhead shared file system with NFS semantics. It provides the best features of P2P storage such as replication, location transparency, and load balancing, which are also the goals of Freeloader. It also provides basic NFS functionality like hierarchical file organization, directory listings and file permissions by builiding it on top of NFS. The main goals of Kosha compare well with those of Freeloader, except that Freeloader does not provide common file system services and Kosha is targeted more towards academic and corporate settings while Freeloader is targeted towards much larger scientific data sets, and acts mostly as a cache space. With Kosha, performance is measured in terms of overhead incurred for common file manipulation commands such as copy, mkdir, grep and stat. In the case of Freeloader, a greater focus is placed on harnessing idle disk bandwidth of workstations and maximizing client-side retrieval throughput.

OceanStore [12] is a global persistent data store designed to scale to billions of users. It provides a consistent, highly-available, and durable storage utility on top an infrastructure comprised of untrusted servers. It is similar to Freeloader in that it is not a file system. Its main contribution appears to be the proposal of multiple groups of storage service providers. Any computer can join the infrastructure and provide storage in exchange for monetary compensation by subscribing to a service provider. Providers negotiate and trade coverage and capacity amongst themselves, providing a quality of service superior to any one company. OceanStore caches data promiscuously; any server may create a local replica of any data object. These local replicas provide faster access and robustness to network partitions. They also reduce network congestion by localizing access traffic. Redundancy and cryptographic techniques are used to protect data from the servers on which the data resides. The main contributions of this project which are relevant to the Freeloader manager are the introspective method used for ensuring locality and the 2-tiered approach used in routing messages. While this project is closely related to the Farsite project, the Farsite project does not address a wide-area infrastructure.

A large number of parallel file systems have also been used in scientific facilites.

Examples of these are GPFS (General Parallel File System) [33], Lustre [8] and PVFS (Parallel Virtual File System) [36, 4]. These file systems are well suited to scientific applications since they use both striping and parallel I/O, and are capable of sustained high retrieval rates. Freeloader is also expected to fit into such a setting by complementing these existing systems. Freeloader serves to fully utilize free space on LAN workstations using a scavenging approach. It is also expected to serve the purpose of a scratch space for intermediate results. Additionally, it reduces the load on existing parallel file systems. With data being aggressively replicated and distributed, it serves as a highly available source of frequently accessed data sets with the ability to deliver sustained throughputs.

Many P2P systems perform the same functions as Freeloader in terms of creating distributed storage infrastructures for file sharing. Some examples are Gnutella [25], Kazaa [26], PAST [11] and BitTorrent [9]. The Freenet project [7] is an adaptive P2P application which pools together space contributions from different nodes to host shared data files. Files are replicated based on their access frequency and proximity to the nodes making the requests. They are also deleted from locations where they are in less demand. Future versions of Freeloader which implement relocation and replication, may draw from ideas used in Freenet. The main contribution of Freenet is that it uses cryptographic techniques to maintain the anonymity of different parties involved in file transfers. Squirrel [22] is a P2P web cache used for exploiting locality in web accesses by sharing the browser caches of different desktops. When a reference is made to a web object contained in this shared cache, it is retrieved from the cache instead of the original data source (web server).

The use of LRU-K cache replacement policy was motivated by the paper [28]. The paper mainly deals with using the policy for database disk buffering. The cache dealt with in the original paper is a buffer of popular disk pages. The replacement policy estimates inter-arrival time based on the total time taken for the last K references. Another paper [27] by the same authors proves the optimality of LRU-K among all algorithms which consider the past K references using Bayesian statistics. In terms of general theory behind page replacement, 2 papers [2, 14] serve as good references.

Chapter 7

Conclusions and Future Work

This thesis presented the manager component of the Freeloader project. An overview of the goals and the architecture of the Freeloader project were presented. We have also discussed the design of the manager and the client. Results indicate that the Freeloader is a viable option for staging Gigabytes of data. Striping has proved to be very effective in balancing load, exploiting parallel I/O, and optimizing space utilization on the Freeloader cloud. The best performance of Freeloader is clearly better than that of GridFTP access to GPFS and HPSS retrieval. The manager's role in terms of registration of benefactors and being a lightweight information server were discussed. A cache eviction policy suitable for large, repeatedly accessed data sets was also presented.

Future work on the manager will focus on accounting for heterogeneous benefactors. This involves striping intelligently according to the capabilities and the load of the benefactors. Striping parameters - stripe width and stripe size will be chosen on-the-fly by the manager for each new data set. We also plan to develop a cost model which effectively captures the relative priorities of data sets and morsels. This would be used in the both the replication and relocation policies. Using a common cost model would also help to integrate these policies with the cache eviction policy. We also plan to develop predictive strategies which help to "brand" or classify benefactors based on their performance histories, to help make better data placement decisions. We are also researching potential optimizations in the area of asymmetric striping, where greater stripe sizes are assigned to nodes closer to the client machine. Security features will need to be incorporated before people are willing to donate their workstations as benefactors. Work on the benefactor side mainly involves finding throttling techniques to minimize the impact on native workload [34].

Bibliography

[1] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J.Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[2] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. In *Journal of the ACM (JACM)*, volume 18, pages 80–93, Jan 1971.

[3] A. Butt, T. Johnson, Y. Zheng, and Y. Hu. Kosha: A peer-to-peer enhancement for the network file system. In *Proceedings of Supercomputing*, 2004.

[4] P. Carns, W. Ligon III, R. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

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

[6] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63(5), 2003.

[7] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2000.

[8] Cluster File Systems, Inc. Lustre: A scalable, high-performance file system. http://www.lustre.org/docs/whitepaper.pdf, 2002.

[9] B. Cohen. Incentives Build Robustness in BitTorrent. 2003.

[10] J. Douceur and W. Bolosky. A large-scale study of file-system contents. In *Proceedings of SIGMETRICS*, 1999.

[11] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, 2001.

[12] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[13] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proceedings of the 19th Symposium on Operating Systems Principles*, 2003.

[14] J. Gray and F. Putzolu. The five minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for cpu time. In *Proceedings of the 1987 ACM SIGMOD Conference*, pages 395–398, 1987.

[15] J. Gray and A. S. Szalay. Scientific Data Federation. In I. Foster and C. Kesselman, editors, *The Grid 2: Blueprint for a New Computing Infrastructure*, pages 95–108, 2003.

[16] A. Gupta, B. Lin, and P. Dinda. Measuring and understanding user comfort with resource borrowing. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, 2004.

[17] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. In *ACM Transactions on Computer Systems*, pages 51–81, Feb 1988.

[18] http://www.hpss-collaboration.org. *High Performance Storage System.*

[19] http://www.ncbi.nlm.nih.gov. *National Center for Biotechnology Information.*

[20] http://www.ncbi.nlm.nih.gov/Genbank/. *GenBank Overview.*

[21] A. Iamnitchi, M. Ripeanu, and I. Foster. Small-world file-sharing communities. In *Infocom*, 2004.

[22] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, 2002.

[23] Lamport, Shostak, and Pease. The byzantine generals problem. In *Advances in Ultra-Dependable Distributed Systems, N. Suri, C. J. Walter, and M. M. Hugue (Eds.), IEEE Computer Society Press*. 1995.

[24] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988.

[25] E. Markatos. Tracing a large-scale peer to peer system: An hour in the life of gnutella. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[26] SHARMAN NETWORKS. The kazaa media desktop. http://www.kazaa.com.

[27] E. O'Neil, P. O'Neil, and G. Weikum. An optimality proof of the lru-k page replacement algorithm, 1999.

[28] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. pages 297–306, 1993.

[29] E. J. Otoo, D. Rotem, and A. Romosan. Optimal file-bundle caching algorithms for data-grids. In *Proceedings of Supercomputing*, 2004.

[30] G. Popek and B. J. Walker. *The LOCUS Distributed System Architecture*. MIT Press, 1985.

[31] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *Proceedings of USENIX Summer Technical Conference*, June 1985.

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

[33] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proceedings of the First Conference on File and Storage Technologies*, 2002.

[34] J. Strickland, V. Freeh, X. Ma, and S. Vazhkudai. Governor: Autonomic throttling for aggressive idle resource scavenging. In *to appear in the 2nd IEEE International Conference on Autonomic Computing (ICAC)*, 2005.

[35] S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, and S. Scott. Freeloader:scavenging desktop storage resources for bulk, transient data. In *Submitted for conference publication*, 2005.

[36] Y. Zhu, H. Jiang, X. Qin, D. Feng, and D. Swanson. Design, implementation, and performance evaluation of a cost-effective fault-tolerant parallel virtual file system. In *Proceeding of the International Workshop on Storage Network Architecture and Parallel I/Os*, 2003.