The Case for Sparse Files

Douglas Thain and Miron Livny
University of Wisconsin, Computer Sciences Department
E-mail: thain,miron@cs.wisc.edu

Abstract

Data intensive distributed applications require precise interactions between storage and computation. The usual abstraction of an ordinary file in a file system does not meet these demands. We propose an alternative: the sparse file. We describe the interface and semantics of sparse files and give examples of how they may be applied to improve the reliability and flexibility of distributed systems. We demonstrate that sparse files may be implemented both easily and efficiently at user level without special privileges. We conclude with a brief demonstration of a sparse file mediating the interaction between storage and computation on behalf of an unmodified scientific application. [1]

1. Introduction

Fewer concepts may be more deeply entrenched in the modern computer system than the idea of a *file*. Today, no matter what the operating system, storage technology, or file system in use, a file is nearly unanimously defined as a simple byte array with a known length and accessed through a small set of well-known operations.

Yet, the file is more than just an interface to a storage system. In a traditional operating system kernel, the file is also the focal point of interaction between the otherwise autonomous systems that mediate access to CPUs and I/O devices. As a process issues operations on a file, the scheduler may remove it from the CPU and place it in a wait state. As file operations complete, a process may regain a CPU, though not perhaps the one it initially had.

This subtlety is not present in most distributed systems. Software engineering principles dictate modularilt, so many significant and powerful systems address solely the problem of managing CPUs or I/O systems, but rarely address the interaction between the two. Nowhere is this separation more complete than in the field of *grid computing*. This branch of computer science aims to provide ready access to large amounts of computing power primarily for solving scientific problems of unprecedented scale.

A wide variety of mature production systems for managing a grid of CPUs are deployed around the world. Systems such as Condor [17], LSF [27], PBS [11], and LoadLeveler [1] provide queueing, scheduling, accounting, and fault tolerance for users that have massive amounts of CPU-intensive jobs. However, these systems have very limited I/O capabilities. Condor, for example, treats remote I/O as an inexpensive task that may be performed while holding a allocated CPU idle.

A number of research and production systems are exploring the problem of handling large amounts of distributed data. The Globus replica manager [26] seeks to duplicate shared datafiles near to where CPUs will require them. The LBL Storage Resource Manager (SRM) [21] exports disks and tapes as allocable and schedulable resources. The Fermi Sequential Access Manager (SAM) [18] seeks to maximize the efficiency of a shared tape archive. Each of these assumes CPU power is easily harnessed on demand as data becomes available.

But how may the two meet?

The only interaction available to users today is alternate sequential allocation. One must extract input files wholly from an I/O system into temporary storage, then make use of the CPU system, and perform I/O again once after the CPU system has completed. The problem with this approach is that it prevents the overlap of potentially independent resources, Further, the private scheduling policies of each system may introduce unexpected latencies, leaving both idle to the user. A better system would allow both systems to be accessed independently and possibly simultaneously, with an automatic mechanism for synchronizing when necessary.

We propose a return to the model present in the classic operating system. We introduce the *sparse file* as a synchronization device between I/O and CPU subsystems. Like traditional data structures such as the sparse array or sparse matrix, the sparse file allows arbitrary portions of itself to be empty. Unlike a plain file, it exposes what por-

[1] This research was supported in part by a Cisco Distinguished Graduate Fellowship and a Lawrence Landweber NCR Fellowship in Distributed Systems.

tions of itself are available to be processed. The sparse file serves as a named rendezvous between any combination of jobs and grid subsystems. This rendezvous introduces flexibility in allocation. Subsystems may be allocated independently and simultaneously, thus allowing overlap for improved throughput while still enforcing synchronization when necessary. The sparse file also fills other useful roles when knowledge of the completion of a file is necessary for a reliable system.

In this paper, we explore how sparse files may be used to couple existing systems together. We begin by describing exactly what a sparse file is and how it works. We then discuss how four common distributed computing structures may make use of the sparse file. We present a user-level sparse file server that is simple and provides reasonable performance for distributed applications. We also demonstrate efficient ways of accessing sparse files from applications without rewriting, rebuilding, or applying special privileges. Finally, we give a short practical example of a sparse file as a synchronization device in a working system.

2. Environment

The concept of a sparse file is most useful in large, large, loosely-coupled, wide-area systems. Such systems have been known by various names over the years, including distributed systems, meta-computers peer-to-peer systems, and most recently, grids. We will use the latter term. Such systems are unique not because of their performance properties, but in the high degree of independence of their components. We are targeting systems with these characteristics:

Distributed ownership. Every component in a grid, ranging from clients to servers and the network in between, may conceivably have a different owner. No global scheduling policy could possibly satisfy the needs of all parties simultaneously, nor does there exist any device for implementing such a policy. Although some components may accept reservations and issue schedules, each interaction ultimately occurs according to the temporary and explicit consent of the principals involved.

Frequent failure. As the number of devices in a system scales, so does the frequency of failure. Networks suffer outages, machines are rebooted, and programs crash. Owners of resources may offer their hardware for public consumption and then suddenly retract it when their private use. Processes executing in such a system are frequently placed on and evicted from several CPUs before they finally execute to completion. Data transfers make suffer multiple disconnections and other failures before running to completion.

Uncertain delays. As a changing community of users clamors for service from a decentralized, fault-prone service, few guarantees on service time can be made. Properties ranging from process queueing time to data transfer time will vary wildy with the many variables in the system.

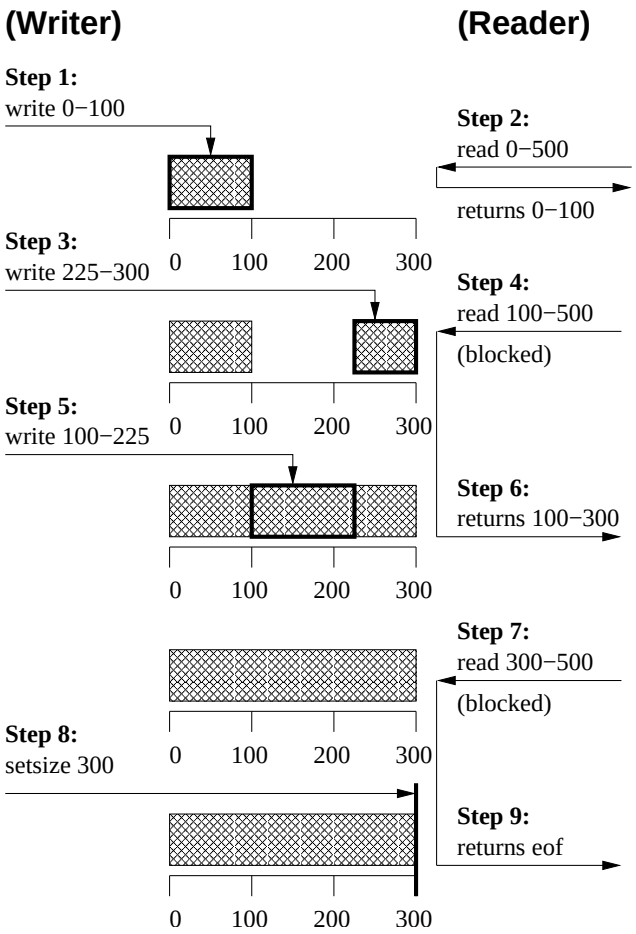

Figure 1. A Sparse File

Transience. Few items in such a system will be truly permanent. Much of the complication necessary to use the system lies in the tasks surrounding the actual work to be done. For example, programs do not simply execute on a CPU; they must discover, claim, activate, and release it in a well-specified way. Likewise, data files are not simply present or absent in the system, but move through a lifetime; they are first expected, then in transit, then arrived, and finally evicted.

This is the environment that hase driven our need for sparse files. Let us proceed to describe exactly what a sparse file is, and then explain how it can put to use to address these problems.

3. Sparse Files

A sparse file is structured container for data storage and process synchronization.

Like a plain file, it provides a sequentially-numbered array of storage bytes. Any position in the array may be empty or may contain a single byte. A *range* is a consecutive set of positions in the array which may or may not be filled. A range of filled bytes is known as an *extent*, while a range of empty bytes is known as a *hole*, in accordance with convention.

Unlike a plain file, holes are not assumed to be filled with a default value. An attempt to read from a hole will result in a delay or an explicit **timeout**, as described below. Also in contrast to a plain file, a sparse file has a size attribute that is set and queried independently of the file data. A sparse file may contain several extents and yet have no size attribute. Or, it may have a size attribute set in a position where no extents exist.

The presence or absence of data may be used for process synchronization. If a process attempts to read a range in the file, several things may happen. If the range begins with an extent, the overlapping portion will be returned to the process. If the range does not begin with an extent, then the process will be blocked until a writing process provides one. The writing of an extent has no bearing on the file's size attribute. A reader will not be informed of the end of a file until a writer explicitly provides one.

Finally, every interaction with the sparse file is limited by a timeout. This simple escape clause allows a client to control the interaction between CPU and I/O access. In addition, both readers and writers may be blocked due to any implementation-specific concerns such as disconnected storage. We will see examples of this below.

Figure 1 is an example of two processes interacting through a sparse file. On the left side, a writer process writes three extents into the file and then sets the logical file size. On the right side, a reader process attempts to read the file sequentially, and is eventually satisfied over the course of several operations.

Here's how the example works:

(1) The writer begins by writing 100 bytes to the beginning of the file. (2) The reader, not knowing how much is available, attempts to read bytes 0-500. Only the available bytes between 0 and 100 are returned. (3) The writer adds another 75 bytes to the file, this time at position 225. This leaves a hole between positions 100 and 225. (4) Again, the reader knows nothing about the rest of the file, and simply attempts to read bytes 100 through 500. There is a hole starting at position 100, so the reader is blocked. (5) The writer fills the hole by writing 125 bytes starting at position 100. (6) This action un-blocks the reader, who then receives all the data between positions 100 and 300. (7) The reader continues to request the data between positions 300 and 500. Note that the reader does not know how "big" the file is, because it has no logical size yet. The reader is blocked because the range after 300 appears to be a hole. (8) The writer completes the file by explicitly setting its size at byte 300. (9) Finally, the reader is unblocked and is given an indication that the end of the file has been reached.

4. Practicalities

Figure 2 shows a concrete interface to a sparse file service. It borrows from several existing interface designs. Three commands closely follow the Internet Backplane Protocol (IBP) [19] approach to name and space management. The remaining eight commands manipulate sparse files directly and are more closely related to the POSIX [14] philosophy of file management.

Three commands manage the namespace of the sparse file service. The **create** command creates a new sparse file and returns a name chosen by the service. The file is issued with a *lease*. [10] The service will destroy the file after the lease time has expired. However, it will never re-issue the same file name twice. Once generated, a file is unique for the lifetime of the service. A client may propose a lease time as an argument to **create**, but the service may revise the actual lease time downward. Before the lease expires, the client may attempt to **renew** it or **delete** it. The service is free to implement any sort of policy limiting the renewal of a file.

The **open** command looks up a file by name and returns a file descriptor (fd) to be used to access the file temporarily. **close** discards this fd. **read** and **write** access an open file according to the sparse file semantics. **setsize** fixes the logical size of a file. **status** gives the positions of any extents in the file and its logical size, if set. **wait** may be used to block the caller until the file has a logical size and an extent that fills it.

The behavior of this interface in ordinary situations should be quite obvious to any programmer familiar with plain files. However, the complexity in both the design and use of most distributed systems lies in the handling unusual situations. Let us shed some light on the consequences of mishaps, errors, (and possibly malice) with respect to sparse files.

According to our philosophy of error management [24], error interfaces must be concise and finite. We have condensed the wide variety of possible error modes in the sparse file interface into four possibilities shown in Figure 3. The **auth** error indicates that the client, however authenticated, is not authorized to perform the requested action. The **name** error indicates that the client has specified a malformed or nonexistent name, such as a bad network address. The **space** error indicates that there is no more storage available, but the client is free to try again later. The **timeout** error indicates that, for any other internal reason, the command couldn't be completed in the time specified by the client.

Command	Arguments	Results
create	lifetime, timeout	name, lifetime
delete	name, timeout	
renew	name, lifetime, timeout	lifetime
open	name, timeout	fd
close	fd, timeout	
read	fd, length, offset, timeout	data, length
write	fd, data, length, offset, timeout	length
setsize	fd, size, timeout	
status	fd, timeout	ranges, [size]
wait	fd, timeout	
commit	fd, timeout	

Figure 2. Sparse File Interface

Error Type	Description
auth	Not authorized to do that.
name	No file by that name (anymore.)
space	Not enough storage space to complete.
timeout	Couldn't complete in the time requested.

Figure 3. Error Interface

There exist a myriad of other error possibilities as diverse as the computer systems available to us. We make no attempt to catalog any sort of error specific to a machine or system, such as a lack of file descriptors or an inability to access internal storage. If a server suffers some internal problem that cannot be resolved by time and cannot be expressed in the standard error interface, then the server is obliged to terminate the client's connection and take any internal recovery or error reporting action that it deems necessary.

All commands are issued in the context of a *session* such as a TCP connection. Although not expressed explicitly in Figure 2, the session is intimately connected with the semantics of the interface. The data-writing commands, **write** and **setsize**, not atomic and must be consummated with a **commit** command, which blocks until all changes are forced to stable storage. If the session is broken – i.e. the connection is lost or the process is killed – then some, none, or all of the requested changes may be visible to later clients. A robust client may return and reattempt any uncommitted actions without knowing whether they have completed successfully. The namespace commands **create**, **delete**, and **renew** are atomic are immediately persistent if successful.

Unlike POSIX, an **open** file is not locked. That is, a file may be **deleted** while open. In this case, any attempts to access a file desciptor referring to a deleted file return the error **name**. The open file descriptor is retained in this invalid state until the client issues a **close**. We chose this behavior in order to afford the owner of a service the maximum possible control to reclaim space from remote users. If we had chosen the POSIX semantics, which require a file to be deleted lazily at the last close, this would allow a remote user to hold storage indefinitely contrary to the owner's wishes.

The timeouts used in the sparse file interface should be should be considered a garbage collection mechanism and not a precision timing tool. The unbounded delays present in every production operating system, storage device, and communications network prevent any other possible interpretation. A **read** that should time out in five seconds may be delayed by fifty seconds if the server process itself is swapped out to backing store. The reply to a **create** indicating a lifetime of one minute could be delayed for several minutes in a congested TCP stream.

For these reasons, the sparse file interface guarantees that a name is never issued twice. Regardless of the many ways that a client and server may disagree on matters of timing, a client will never accidentally access the wrong file. However, a client must always be prepared to deal with a file that returns the **name** error, indicating it no longer exists. A side effect of this requirement is that it (sadly) permits the owner of a storage device to break a valid lease and delete a file before its lifetime has expired. Although this sort of unfriendly behavior will certainly have social consequences, it does not violate the sparse file interface. Clients must be prepared to encounter a **name** error, whatever its source.

An even worse situation may occur if the backing store

for a sparse file server is irrevocably lost, thus breaking the mechanism for generating unique names. A client holding a file name issued by the old server might attempt to present it to a new server established at the old address. If the namespaces of the new and old servers overlap, the client might silently access a file with the same name but different data.

The are several insurance policies against this problem. One is to make use of a highly-reliable, centralized, unique name generator shared by all such file servers. This has the benefit of guaranteed uniqeness, barring the failure of the central server. A more scalable and practical solution is to generate names by incrementing a persistent counter, and then issuing the resulting number with a random string appended. This is not guaranteed to be unique, but the likelihood of name collision may be made vanishingly small by increasing the length of the random string.

5. Applications

Let's consider how a sparse file service may be put to use in a variety of distributed systems. We will consider four structures that reflect real applications and systems related to grid computing.

In each example, we will use a fairly general model of execution that applies to a variety of existing systems. We assume that the user submits jobs to be done to a process called the *planner*. The planner maintains a persistent job list and is responsible for finding places to execute jobs, monitor their progress, and returning results to the user. The planner submits jobs to a remote *queue*, which assigns jobs to more or more CPUs. The queue may delay the execution of a job indefinitely as it mediates between multiple users according to its local policy. Once placed on a CPU, the job begins execution. We assume that it is common for the job to be evicted from a CPU and returned to the queue, perhaps saving its computation in the form of a checkpoint. We also assume that the job is not customized, but instead coupled with an *adapter*, which converts its standard I/O operations into sparse file commands. In addition, the adapter can communicate with the process queue and ask to release the CPU and re-enter the process queue, perhaps with conditions on its restart. Finally, a *sparse file server* makes a storage device available through the interface we have described above.

We call this model *autonomous scheduling*. It has all of the components of a CPU scheduler in a time-sharing operating system: jobs, queues, buffers, I/O devices, and so on. It also has the same sort of events: jobs are scheduled, preempted, and blocked on I/O. However, there is no global policy that joins the whole system together. Even if it were desirable, it would not be possible. A queue cannot know what I/O devices an adapter uses, nor vice versa.

Thus, every interaction requires the consent of the components involved. An I/O device cannot cause a job to be preempted from its CPU. Rather, the I/O device informs the adapter that a delay is likely, and the adapter decides what to do next. It might choose to access another I/O device, to re-enter the process queue, or simply busy-wait if it believes the delay will be short.

Of course, there is an incentive not to busy-wait excessively. The user that consumes excess CPU while busy-waiting pays for the privilege. CPU consumption is usually accounted for in some fashion, and the consumer who does not pay in the form of money is likely to pay in the form of lost priority.

5.1. Staged Input

As we noted above, grid applications will require interactions between independent process queues and data archives. Because each of these systems may delay the user's request for service indefinitely, we use the sparse file to connect them together. Figure 4 demonstrates how a sparse file may be used as a synchronization between a process queue and a data archive.

To begin, the planner must issue a **create** command at the sparse file server to generate a unique input file name. It then may interact with both the queue and the archive in either order or simultaneously. On the queue side, the planner submits the job along with its adapter and instructions to read the sparse file created in the first step. After some delay, the queue executes the job on a CPU, which then attempts to read from the sparse file server. On the archive side, the planner requests the delivery of the needed input file. After another delay, the archive begins to transfer the file into the named sparse file using the **write** and **setsize** commands.

Functionally speaking, it doesn't matter whether the data transfer completes before the job begins to execute or vice versa. If the job should require input data from the sparse file before it is available, it will simply block. If a portion of the file is available, the job may read and process it without waiting for the entire transfer to complete.

Most importantly, the adapter may interact with the queue to manage its CPU consumption in light of the developing I/O situation. If the job makes progress using the first half of the input file, and then discovers the second half is not yet available, the adapter may request to return to the queue and be rescheduled at a later time, perhaps when the whole file has arrived. Just as in a traditional CPU scheduler, the job is not aware of such technicalities. The policy is present in the adapter, which may state something like:

> *If I am stuck more than one minute waiting for I/O, return to the process queue for five minutes.*

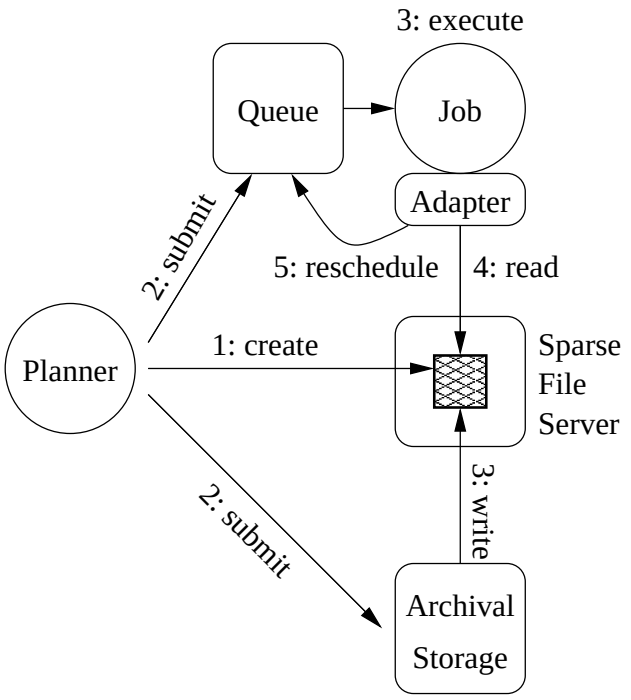

Figure 4. Staged Input

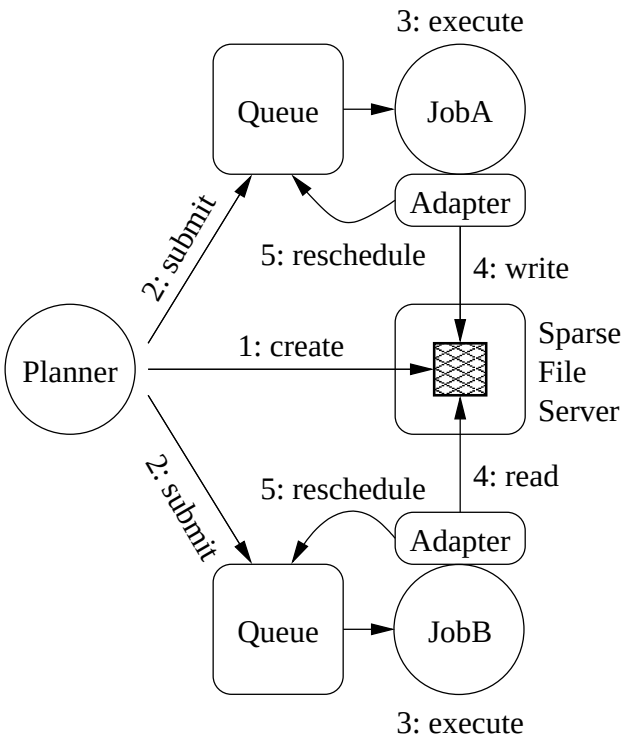

Figure 5. Pipeline

Upon rescheduling, the adapter may reconsider the situation and continue to work or return to the queue once again.

If the input file should be deleted, whether due to an expired lease or a deliberate eviction by the owner, the adapter must exit, indicating to the queue that the process cannot execute at all. The archive, upon discovering that the target name no longer exists, will also abort the transfer. A suitable message is returned to the planner, which may restart the whole process or inform the user as necessary.

Without the sparse file server, we may accomplish something similar, but with far less flexibility. If the sparse file server is replaced by a plain file server, then the entire data transfer must be completed before the job is even submitted. This is because the adapter would not be able to distinguish between a partially-transferred file and a complete file. Further, we may unnecessarily occupy space at the file server, and be charged for occupying it while the job sits idle in the queue.

This is not to say that job submission and data transfer should always be simultaneous. Indeed, if the transfer time is very long and the job queueing time is very short, transferring the whole file before submitting the job its sensible. Rather, the sparse file semantics allow a wide variety of scheduling decisions to be made without relying entirely on performance assumptions that may be unreliable. The job and the data may be queued simultaneously without suffering an incorrect execution when our performance assumptions are violated.

5.2. Pipeline

A significant number of grid computing applications are conceived as a pipeline of processes, each reading its input from the output of the previous stage. An example of such a pipeline is the series of computation steps required by the Compact Muon Solenoid (CMS), [12] a high-energy physics experiment in preparation at CERN. CMS connects several distinct software packages for the generation of random events, the simulation of the detector, and the reconstruction of raw data, whether simulated or measured.

Figure 5 shows how a pipeline may be constructed using a sparse file. As before, the planner creates a file at the sparse file server. It then submits two jobs, perhaps to two different queues. The jobs may then begin to execute in any order. As job A produces output, it writes it to the sparse file. As such data becomes available, job B is unblocked and may read and process it. Either process may ask to be rescheduled if it is blocked waiting to access the sparse file.

This service may seem remarkably like the standard pipe abstraction. However, there are two key differences. First, the sparse file permits random access, so it may be used even when the interaction between the processes is not sequential. This is common between applications not specifically designed for a distributed system. Second, a sparse file does not equate a disconnection with an end of file condition. This property makes this construction robust to the

common errors that plague a distributed system. If the network should fail, both processes may reconnect and continue to work without harm. If one process is evicted or killed, it may resume from its last checkpoint without harming the other.

The drawback is that the sparse file contains the entire history of the interaction. Thus, it is not suitable as a connection between processes that run indefinitely or exchange more data than can be stored.

5.3. Buffered Output

A number of systems have recognized the utility of a *buffer server* [5] that provides fast access to nearby storage while delivering data asynchronously to another target. Examples of this idea include the Trapeze buffer server, the WayStation [16], and the Kangaroo system. [22]

The complication of using a buffer server is the problem of reconciliation. How much time must a process wait to let the buffer finish its work? Will it know if the buffer fails? How can it retract work that relies on a failed buffer? These questions are usually answered unsatisfactorily in one of two ways:

1. *The writer must trust the buffer to complete reliably and in bounded time.* This assumption seems unwise in a grid environment where storage may be borrowed from a third party and networks and systems regularly fail.

2. *The writer must block until the buffer has finished.* This solution is reliable. If the buffer should fail, so will the writer, and the whole system may be re-tried. However, this also forces the caller to waste a CPU allocation by holding it idle while waiting for the buffer to complete its work. This is exactly the problem a buffer server is designed to solve, so this answer is counter productive.

A better solution is built using a sparse file, as shown in Figure 6. A job executes via a remote queue as in the other examples. However, it writes its output as a series of messages passed to a nearby buffer server. These messages include **write** commands which carry file data as well as any **setsize** commands issued when the job completes. Once complete, the job may exit the queue and return to the planner without worry about the buffer in any way. Asynchronously, the buffer sends the file updates back to a sparse file server containing a file created by the planner. The planner may watch the file with the **wait** command and detect when all the messages have arrived. It may then indicate to the user that the job has completed.

The buffer is free to delay or reorder packets as much as it wishes. The user must simply inform the planner how

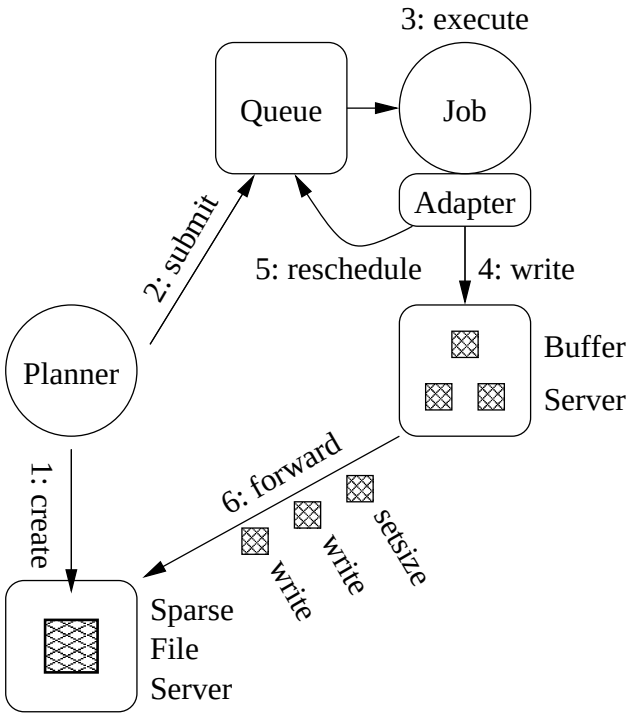

Figure 6. Buffered Output

long she is willing to wait for the results before attempting the job again. Of course, a re-submitted job must be assigned to direct its output to a different sparse file, lest the output of two runs be confused. Even if a faulty buffer discards messages, a penalty is paid only in performance. The planner is capable of telling when the buffered operations have completed.

5.4. Audited Output

Most applications that create output destined for an archive requires a certain degree of auditing. That is, their output must be checked by a human or a program for legitimacy or value before being added to a permanent archive. Such auditing may take place when the output is completed, but it is not uncommon to also happen as the application runs.

An example of auditing takes place at the National Center for Supercomputing Applications (NCSA) in Champaign, Illinois. Here, researchers submit batch jobs to run the Gaussian [9] atomic simulation application. Gaussian generates a very large log output as it runs over the course of hours or days. By examining the log file as it is produced, researchers may identify useless runs (i.e., divergent results implying bad parameters) early in the execution of the job. Such runs may be cancelled to avoid wasting computing resources. Likewise, the output may be evaluated

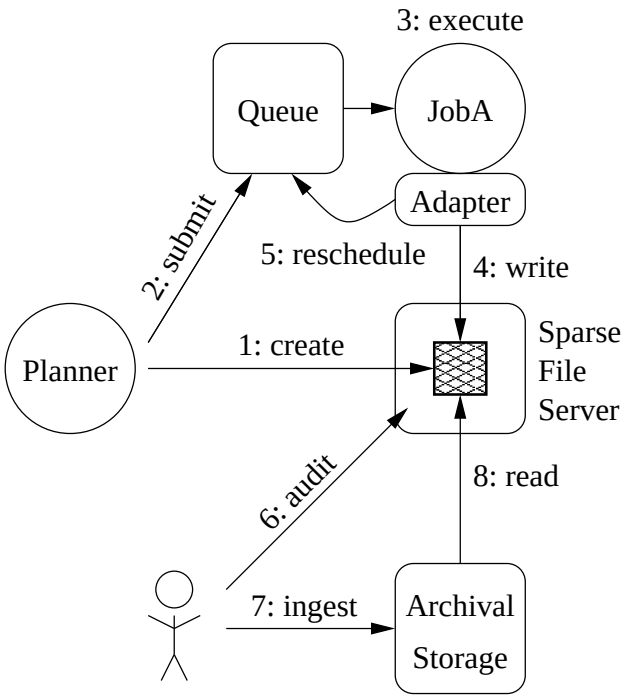

Figure 7. Audited Output

after the job completes before deciding whether to place it in archival storage.

Auditing is currently implemented informally. The job writes its output to the local disk of its execution site. Another process [2] periodically copies the output to a mass storage system, when then may automatically move the (possibly incomplete) output file to archival storage during periodic backups. Although this system is acceptable for small-scale use, it consumes twice the storage necessary, does not permit the job to migrate, potentially archives invalid output files.

Auditing may be implemented easily and efficiently without duplication of storage by using a sparse file as the synchronization point between three parties: the application, the user, and the archive. This is shown in Figure 7. The job executes as in previous figures and may migrate and rescheduled as necessary. As the output is produced to the sparse file, a human (or program) may audit the output at leisure by using the sparse file interface. The completion of the output is detected when the job sets the logical file size. After the job completes, the sparse file may remain according to the constraints of its lease. If the user accepts the output as valid, it may direct the archival system to ingest the output file. If the user rejects or simply forgets to audit the file, it will eventually be reclaimed by the sparse file server.

[2] This process is known colloquially as the "grid-enabled cat", referring to the simple UNIX program of the same name.

6. Implementation

We have argued that the sparse file model simplifies the construction of a variety of distributed systems. Next, we will demonstrate that sparse files may be easily implemented at user level with no special privileges. This implementation gives reasonable performance within certain bounds, although we will point out its performance limitations and leave its optimization open for future work.

6.1. Server

We have built a sparse file library and server in user space on top of a standard POSIX operating system and filesystem. This permits it to be deployed on arbitrary machines without special privilege or kernel modifications. We make no claim that this implementation gives optimal performance. Instead, we intend to show that a simple implementation exists and that the proper semantics can be provided with reasonable performance under expected loads.

In order to implement the sparse file semantics correctly, we must carefully examine the semantics of the underlying system. The various and sundry implementations of the POSIX interface vary greatly in how they guarantee data integrity after a crash. Some, like NFS [20] guarantee the atomic persistence of every individual operation. Others, like AFS [13] make no guarantees until a file is successfully closed. Local file systems such differ in the narrowest of subleties such as the ordering of distinct write operations. In order to give the correct semantics in all of these environments, we build our system using very conservative assumptions that we believe to hold universally:

1. All updates to a single file are suspect until a successful **fsync** and **close**.

2. **rename** may be used to atomically replace one file with another.

With these assumptions, we may implement a sparse file in terms of two standard POSIX files, shown in Figure 8. One standard file (`data`) contains the literal sparse file data with a one-to-one mapping from sparse file offsets to physical file offsets. Explicit holes in the sparse file are represented as implicit zero-filled holes in the physical file. The second file (`meta`) contains all of the sparse file's meta-data, such as the logical file size and a list of filled regions in the file.

When updating the data in the sparse file, new data is simply written into `data` using the standard POSIX I/O commands. The meta-data, including a linked list of extents, is kept in memory. To implement **commit**, four steps are necessary:

1. **fsync** is called on `data` to force all newly-written data to stable storage.
2. A new file (`meta.tmp`) is created and filled with the updated meta-data.
3. **fsync** is called on `meta.tmp` to ensure it is on stable storage.
4. **rename** is called to atomically replace `meta` with `meta.tmp`.

If a crash should occur before these four steps complete, then some portion of the new data may be in stable storage, but without the updated `meta`, such areas will still be perceived as holes. If the user is writing over an existing extent, then such ranges may or may not reflect the new data, which is still consistent with the sparse file semantics. The atomic **rename** ensures that `meta` is never found in a partially reconstructed state.

6.2. Performance

Our primary concern for the performance of the sparse file is that it provides sustained throughput competetive with plain files. This is an inflexible requirement necessary to interact with high-bandwidth grid CPU and I/O managers. The latency of individual operations is less of a concern, as they are likely to be dwarfed by the latencies of operating in a wide area network.

We compared the throughput of the sparse file library against that of plain files on a stock workstation. The experimental machine had a MSI 694D Pro-AR motherboard with a VIX 694X Chipset, two 933MHz Pentium-III CPUs, and 1 GB SDRAM. The I/O subsystem included a Promise ATA 100 IDE disk controller, and a 30GB Ultra ATA 100 IBM 307030 disk. The machine ran Linux 2.4.18 with an EXT3 file system.

We will show the results of writing files to disk. Read performance was largely the same. Each point in the following graphs represents the mean of 25 measurements, less up to five outliers. Outliers were identified as measurements less than 80 percent of the mean, and were found to be due to periodic system tasks such as maintenance and logging. The variance of the remaining data, elided for clarity, were less than ten percent of the mean.

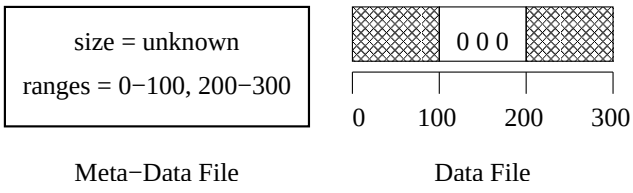

Figure 8. Sparse File Implementation

Figure 9 show the burst performance of writing large files into the system buffer cache. These writes are considered unsafe because they are not followed by a **commit** (for sparse files) or a **fsync** (for plain files) and may be incomplete or corrupted if a system reset occurs between writing the data and re-reading it. Figure 12 shows the same experiment, this time performed safely by completing each transfer with a **commit** or **fsync** as appropriate. In both cases, the sparse file is quite competetive with the plain file. The expense of manipulating the extent list is quite small: there is only one continuously growing extent. Likewise, the extra few synchronous operations needed to implement **commit** are hidden by the primary task of moving the raw data.

Figure 10 demonstrates unsafe random write performance. Here, we wrote 4KB blocks at random offsets into files of increasing size. The total amount of data written was equal to the file size, but the files were heavily fragemented by the random offsets. The sparse file has very poor performance as the number of extents increases. A similar situation is seen in Figure 13, which shows safe random writes. We may remedy the situation by increasing the block size modestly. Figure 11 show the same experiment with a block size of 128 KB. The sparse file is measurably slower, but still exceeds the raw bandwidth of the disk. The two types are again undistinguishable in Figure 14, which writes safely and randomly with a 64KB block size.

To use this naive implementation efficiently, we must take care not to use microscopic block sizes when writing randomly. This is not an unreasonable burden when we consider that the other constraints of distributed computing tend to encourage larger block sizes.

6.3. Adapter

Although the enthusiastic programmer might wish to rewrite applications to use the sparse file interface directly, we can hardly expect the vast majority of programs to be modified at the appearance of a new storage system. Likewise, if we aim to use widely distributed systems not directly under our control, we must accept the installed operating systems and cannot deploy kernel changes to support sparse files.

To make the sparse file service useful, we must have a unprivileged adapter to connect existing programs to new storage devices. These devices are sometimes know as *interposition agents*. [15] There are many techniques for accomplishing this, ranging from system call interception to binary rewriting. An excellent review is given by Alexandrov et al. [3]

We have built a general-purpose adapted for connecting standard POSIX applications to a variety of distributed I/O services. [25] This adapter is called the Pluggable File System (PFS) and is shown in Figure 15. PFS speaks a variety of protocols, including Chirp [8], GridFTP [4], HTTP, Kan-

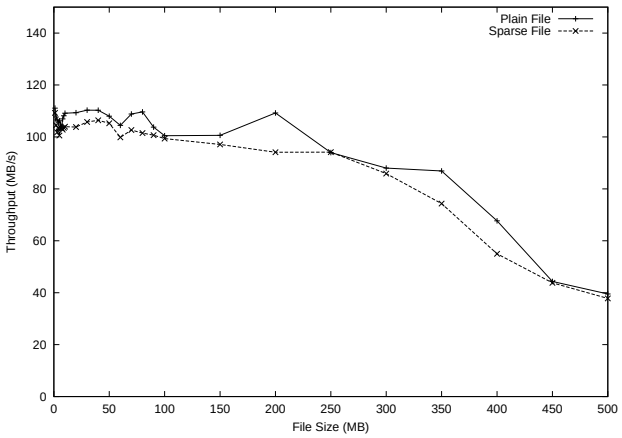

Figure 9. Unsafe Sequential Write

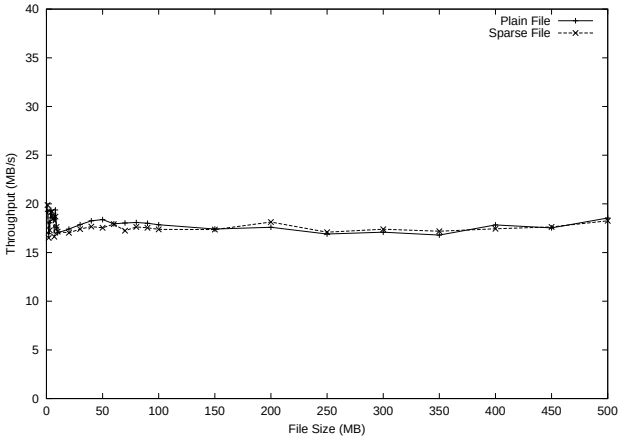

Figure 12. Safe Sequential Write

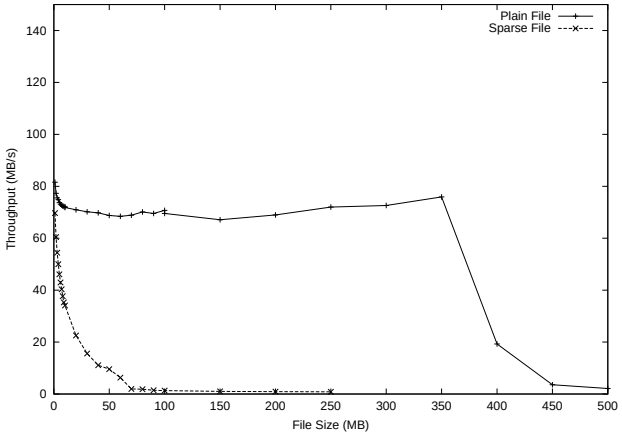

Figure 10. Unsafe Random Write

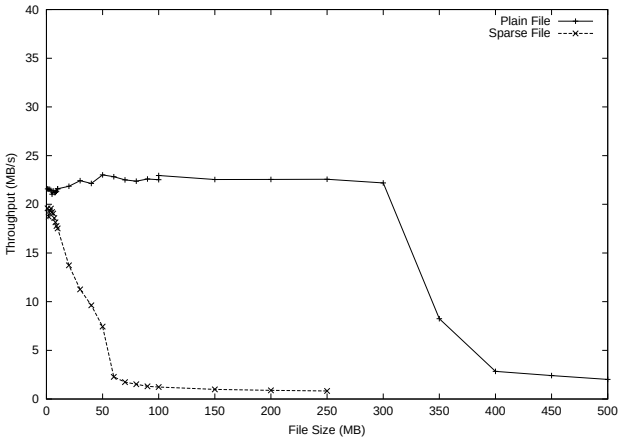

Figure 13. Safe Random Write

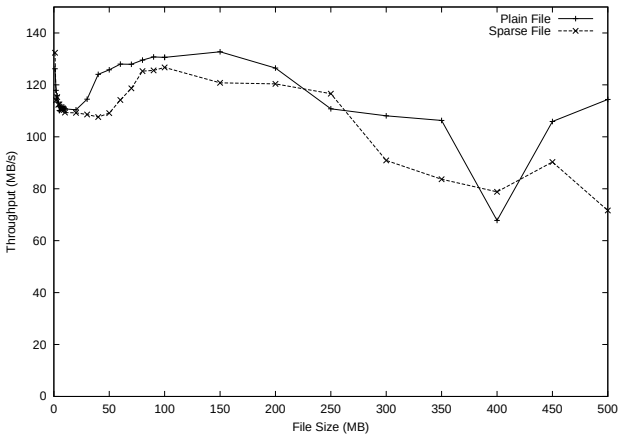

Figure 11. Unsafe Random Write with Large Blocks

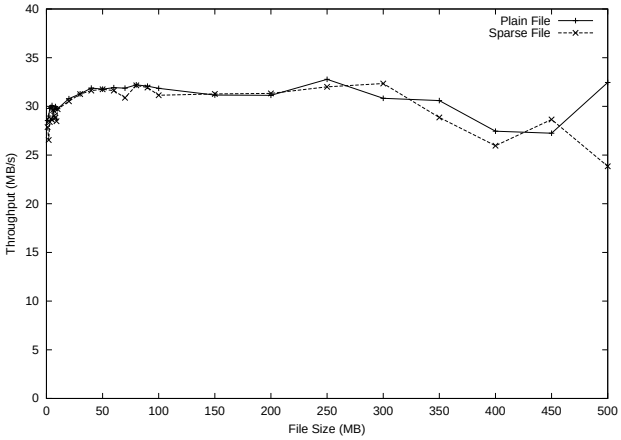

Figure 14. Safe Random Write with Large Blocks

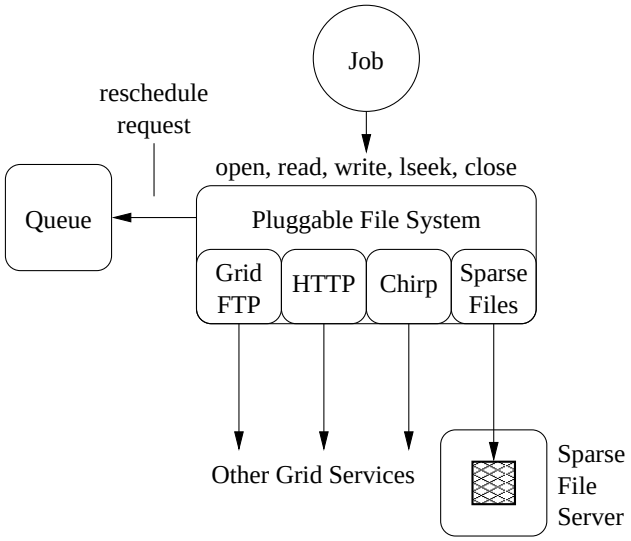

Figure 15. Adapter Detail

garoo [22], SRB [6], and of course the sparse file interface.

PFS is built with Bypass [23], a general tool for building such interposition agents. It is a shared library with the same interface as the standard C library, so it may be applied to any dynamically-linked application that uses ordinary I/O calls. Because the interposition technique involves simple function calls, the overhead of the adapter itself is measured in microseconds [23], which is inconsequential compared to the cost of the network and I/O operations themselves.

PFS is structured much like the I/O system of a standard operating system. For every process, it has a table of open file descriptors, a set of seek pointers, and device drivers that represent each of the remote protocols. Applications may access files in the local system using ordinary path names, or they may explicitly use fully-qualified names in a syntax resembling a URL:

/http/www.yahoo.com/index.html
/ftp/ftp.cs.wisc.edu/RoadMap
/sparse/server.wisc.edu/105-qnvismgy

The job of the adapter does not end with the physical transformation of its I/O into sparse file operations. It must also provide logical transformations to deal with all of the issues that the application is unaware of. This includes a name mapping, a reliability policy, and a scheduling policy.

Although some applications may be instructed to use sparse files by explicitly passing fully-qualified names as arguments, most applications use fixed file names of one kind or another. For example, most programs implicit access `/etc/passwd` to find names in the user database. For better or worse, many scientific applications use *only* hard coded input and output file names. PFS provides a name mapping to support such applications. This is simply a table that converts simple file names into fully-qualified file names:

/etc/passwd	=	/sparse/server.wisc.edu/abc
/tmp/output_data	=	/sparse/server.wisc.edu/def
<stdout>	=	/sparse/server.wisc.edu/ghi

In the structures that we have described above, the planner generates sparse file names as it creates them, and is thus responsible for creating a name mapping for each process.

Because the job is not aware of the vital **setsize** and **commit** sparse file operations, the adapter is responsible for choosing appropriate times to execute them. Because **setsize** is used to indicate the completion of an output file, the adapter only issues it on an output file when the application has successfully run to completion. It cannot perform it any earlier, as some jobs may open and close a file multiple times before they are truly done with it. The **commit** operation is issued when the application requests an **fsync** and also at the end of execution.

The job is also not aware that it is executed by a remote process queue, so the adapter must manager that interaction as well. Although rescheduling policies may be arbitrarily complex, ours is quite simple. The planner may specify a minimum time it is willing to allow the job to hold a CPU idle while it waits for I/O. If this time expires, it also gives a minimum time to wait in the queue before it may be scheduled on another CPU to reconsider the situation. This policy is specified to PFS as follows:

max_io_wait_time	=	60
min_queue_wait_time	=	300

7. Example

We will briefly demonstrate the use of a sparse file as a synchronization device for input data, as shown in Figure 4.

For didactic purposes, we use **corsika**, the first stage of the AMANDA software pipeline. [2] From a physical science perspective, **corsika** generates a shower of neutrinos from a random seed and a large set of parameters describing the physical universe. From a computer systems perspective, **corsika** reads about 2.6 MB from a set of five input names, and computes for about 100 seconds as it produces about 1.2 MB of output in two output files.

We submit **corsika** to the Condor system at the University of Wisconsin. Using PFS as an adapter, its fixed input and output file names are directed to a nearby sparse file server. In addition, the adapter is given a simple scheduling policy:

> *If I am stuck more than ten seconds waiting for I/O, return to the process queue for thirty seconds.*

While the process is under the control of the Condor system, we simulate a slow data source by hand, transferring in the input files at different times to demonstrate three different sparse file interactions. In Figure 16, the data transfer begins immediately, but the process is delayed in the queue. When it is finally scheduled, the data are available, and it executes immediately. In Figure 17, the job begins executing immediately, but the data transfer is delayed. According to the adapter's policy, the job is activated repeatedly, but seeing no data available after ten seconds, returns to the queue. Once the data start to arrive, the job executes to completion. In Figure 18, the job begins to execute while the data it needs are still in transit. It begins executing slowly, occassionally blocked while waiting for needed input data. As the data transfer outstrips the job's input needs, it begins to speed up.

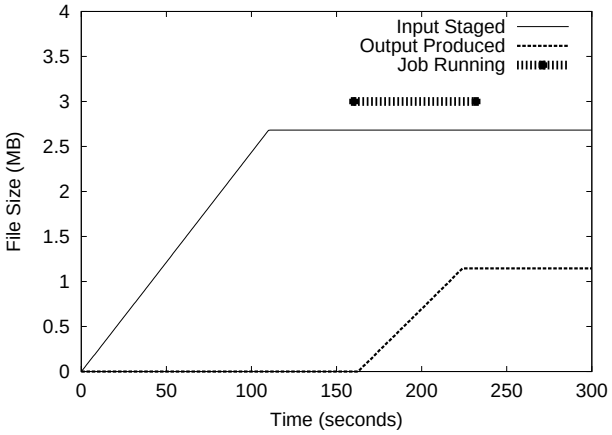

Figure 16. CPU Delayed

8. Related Work

A variety of projects are exploring the design space of fundamental storage devices for distributed computing. These devices form the "building blocks" of distributed storage and are typically managed by a higher-layer system. The minimalist Internet Backplane Protocol (IBP) [19] is an interface to plain files with flexible policies. and controls for space and time management. GridFTP [4] is an extension of the standard File Transfer Protocol (FTP) which emphasizes security and high-performance access to large files. NeST [8] is a single storage device encompassing multiple protocols, space management, replication, security, and quality of service. All of these devices export a plain file abstraction,

The ExNode [7] is higher-level structure built on top of a distributed set of IBP servers. The ExNode is analogous to a standard filesystem inode and provides described by its designers as "something like a file." It logically merges extents spread across a distributed system into a single logical file, possibly with holes or multiple redundant units. Because the devices underlying the ExNode are physically remote, no portion of the ExNode is guaranteed to be available at any given time.

The ExNode and the sparse file both address a similar problem, albeit from different perspectives. The ExNode is a *structure* for aggregating remote resources. The sparse file is an *interface* for communicating with a file. The ExNode, with small modifications, could present a sparse file interface. Or, the ExNode could be built on top of sparse file servers, thus adding a new degree of flexibility (and failure modes) into aggregate distributed storage.

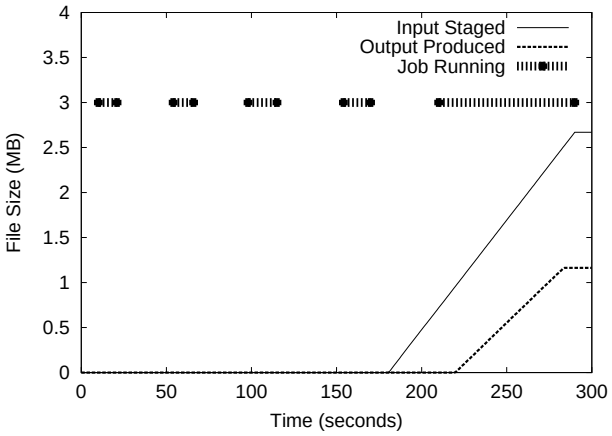

Figure 17. I/O Delayed

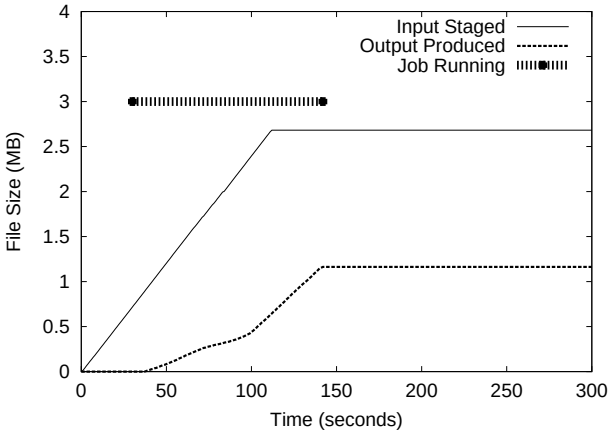

Figure 18. Neither Delayed

9. Conclusion

Conclusion not yet written.

References

[1] *IBM Load Leveler: User's Guide*. I.B.M. Corporation, September 1993.

[2] AMANDA project home page. http://amanda.berkeley.edu, September 2002.

[3] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. UFO: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, pages 207–233, August 1998.

[4] W. Allcock, A. Chervenak, I. Foster, C. Kesselman, and S. Tuecke. Protocols and services for distributed data-intensive science. In *Proceedings of Advanced Computing and Analysis Techniques in Physics Research (ACAT)*, pages 161–163, 2000.

[5] D. Anderson, K. Yocum, and J. Chase. A case for buffer servers. In *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems (HOTOS)*, April 1999.

[6] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of CASCON*, Toronto, Canada, 1998.

[7] M. Beck, T. Moore, and J. Plank. An end-to-end approach to globally scalable network storage. In *ACM SIGCOMM*, Pittsburgh, Pennsylvania, August 2002.

[8] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. A. Dusseau, R. Arpaci-Dusseau, and M. Livny. Flexibility, manageability, and performance in a grid storage appliance. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.

[9] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, V. G. Zakrzewski, J. J. A. Montgomery, R. E. Stratmann, J. C. Burant, S. Dapprich, J. M. Millam, A. D. Daniels, K. N. Kudin, M. C. Strain, O. Farkas, J. Tomasi, V. Barone, M. Cossi, R. Cammi, B. Mennucci, C. Pomelli, C. Adamo, S. Clifford, J. Ochterski, G. A. Petersson, P. Y. Ayala, Q. Cui, K. Morokuma, D. K. Malick, A. D. Rabuck, K. Raghavachari, J. B. Foresman, J. Cioslowski, J. V. Ortiz, A. G. Baboul, B. B. Stefanov, G. Liu, A. Liashenko, P. Piskorz, I. Komaromi, R. Gomperts, R. L. Martin, D. J. Fox, T. Keith, M. A. Al-Laham, C. Y. Peng, A. Nanayakkara, C. Gonzalez, M. Challacombe, P. M. W. Gill, B. Johnson, W. Chen, M. W. Wong, J. L. Andres, C. Gonzalez, M. Head-Gordon, E. S. Replogle, and J. A. Pople. Gaussian 98 revision a.7, 1998.

[10] C. Gray and D. Cheriton. Lease: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Twelfth ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–210, 1989.

[11] R. Henderson and D. Tweten. Portable batch system: External reference specification. Technical report, NASA, Ames Research Center, 1996.

[12] K. Holtman. CMS data grid system overview and requirements. CMS Note 2001/037, CERN, July 2001.

[13] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systmes*, 6(1):51–81, February 1988.

[14] IEEE/ANSI. Portable operating system interface (POSIX): Part 1, system application program interface (API): C language, 1990.

[15] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM symposium on operating systems principles*, pages 80–93, 1993.

[16] M. Kim, L. Cox, and B. Noble. Safety, visibility, and performance in a wide-area file system. In *1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

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

[18] L. Loebel-Carpenter, L. Lueking, C. Moore, R. Pordes, J. Trumbo, S. Veseli, I. Terekhov, M. Vranicar, S. White, and V. White. SAM and the particle physics data grid. In *Proceedings of CHEP*, 1999.

[19] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Proceedings of the Network Storage Symposium*, 1999.

[20] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, 1985.

[21] A. Shoshani, A. Sim, and J. Gu. Storage resource managers: Middleware components for grid storage. In *Proceedings of the Nineteenth IEEE Symposium on Mass Storage Systems*, 2002.

[22] D. Thain, J. Basney, S.-C. Son, and M. Livny. The Kangaroo approach to data movement on the grid. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, San Francisco, California, August 2001.

[23] D. Thain and M. Livny. Multiple bypass: Interposition agents for distributed computing. *Journal of Cluster Computing*, 4:39–47, 2001.

[24] D. Thain and M. Livny. Error scope on a computational grid. In *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC)*, July 2002.

[25] D. Thain and M. Livny. The pluggable file system. Technical Report in preparation, University of Wisconsin, Computer Sciences Department, 2002.

[26] S. Vazhkudai, S. Tuecke, and I. Foster. Replica selection in the globus data grid. *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2001.

[27] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a load sharing facility for large, heterogenous distributed computer systems. *Software: Practice and Experience*, 23(12):1305–1336, December 1993.