Efficient Access to Many Small Files in a Filesystem for Grid Computing

Douglas Thain and Christopher Moretti
University of Notre Dame

Abstract— **Many potential users of grid computing systems have a need to manage large numbers of small files. However, computing and storage grids are generally optimized for the management of large files. As a result, users with small files achieve performance several orders of magnitude worse than possible. Archival tools and custom storage structures can be used to improve small-file performance, but this requires the end user to change the behavior of the application, which is not always practical. To address this problem, we augment the protocol of the Chirp filesystem for grid computing to improve small file performance. We describe in detail how this protocol compares to FTP and NFS, which are widely used in similar situations. In addition, we observe that changes to the system call interface are necessary to invoke the protocol properly. We demonstrate an order-of-magnitude performance improvement over existing protocols for copying files and manipulating large directory trees.**

I. Introduction

Much research in networking and grid computing has focused on the efficient storage, transfer, and management of large files. [1], [2], [3], [4], [5], [6] For many important applications, the quantity of data to be processed exceeds the storage capacity or reasonable transfer times that can be achieved on conventional systems. For these large-data applications, a variety of techniques such as disk aggregation [7], [8], [9], [10], parallel network transfer [11], [12], peer-to-peer distribution [13], and multi-level storage management [2] are essential for constructing usable grid computing systems.

However, there are also many production workloads that manipulate primarily large numbers of small files. For example, in bioinformatics applications such as BLAST [14], it is common to run thousands of small (less than 100 bytes) string queries against a constant genomic database of several GB. In other cases, a standard grid application may depend on the installation of a complex software package consisting of executables, dynamic libraries, and configuration files. The package must either be accessed at run-time over the network, resulting in many small network operations, or installed on a worker node, resulting in a large number of small file creations. Or, a grid computing system may create a large number of small files internally in the course of executing a workload for the inputs, outputs, log files, and so forth.

Unfortunately, the data throughput of small file operations on both networks and filesystems is many orders of magnitude worse than the bulk transfer speeds available with large files. On the network, this is because protocols such as FTP [15] treat a single file as a distinct heavyweight transaction that requires an individual network stream and authentication step. A network filesystem such as NFS [16] has the opposite problem: files are access on demand in small page-sized chunks, resulting in many network round trips and poor performance. This is particularly harmful in grids, where high network latencies are common.

In some sense, the solution is easy: collections of small files should be aggregated into large files and then transferred in bulk. Archival tools can be used to collect many files into a single compressed file. Some grid storage systems, such as the Storage Resource Broker [2] introduce an explicit mechanism called *containers* to encourage grouping for efficient transfer to and from tape. When the set of files is known in advance, many transfers can be pipelined or parallelized as shown by Silberstein [17] and Bresnahan [18].

However, these techniques are not always applicable:

- The user may be generating a sequential workload from a traditional script or interactive tool, so the set of all files is not known to the system in advance.
- When copying archives to a remote file server, there may not be a facility at the receiving end for unpacking the data into multiple files.
- The semantics of the system may demand multiple files: as above, the batch system may insist on having a distinct input and output file for every single job in a workload.
- The user may not be technically inclined, and simply takes an existing program, and multiplies by one thousand instances before submitting it to the grid.
- The user may not even be aware that a system is in fact distributed, and simply expects a large recursive directory copy to proceed with reasonable performance.

Can we have a filesystem for grid computing that supports efficient access to large numbers of small files while still preserving good performance for large files and avoiding any change in user behavior? We demonstrate that it is possible by augmenting the protocol used by Chirp, a distributed filesystem for grid computing, with hybrid operations that combine elements of traditional RPC and streaming I/O. However, a protocol is no use unless programs can employ it. We also demonstrate how user-transparent changes to the system call interface and command line tools can assist programs in harnessing the protocol. We demonstrate the performance benefits of this approach on both file copies and on ordinary scripts that manipulate large directory trees.

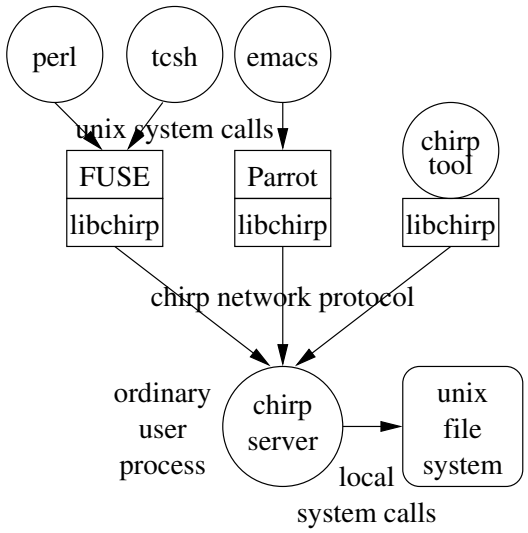

Fig. 1. The Chirp Filesystem for Grid Computing

The Chirp filesystem consists of a user-level server that exports an ordinary Unix filesystem. Users connect unmodified applications to the server using assistive tools such as FUSE and Parrot. The protocol employs standard authentication technologies, allowing for secure wide-area data access.

II. Chirp: A Filesystem for Grid Computing

In previous papers [19], [20], [21], we have introduced Chirp, a filesystem for grid computing. In this section, we offer a brief review of Chirp before describing the new work.

Figure 1 shows the overall structure of Chirp. A Chirp server is a user-level process that runs as an unprivileged user, and exports an ordinary Unix filesystem. The server accepts connections, and then authenticates users via the Grid Security Infrastructure (GSI) [22], Kerberos [23], or by simple hostnames. Access controls are enforced on a per-directory basis using these credentials, allowing users to share data and storage space with highly flexible policies. A hierarchical allocation mechanism allows the storage owner to control how much space is consumed.

Users connect to Chirp in one of several ways. A library `libchirp` is provided that allows custom code to access Chirp servers explicitly. This interface offers the best performance, but we do not generally expect application developers to rewrite their applications to use Chirp. Unmodified applications may be attached to Chirp by employing either Parrot [19] or FUSE [24] to mount the filesystem into the user's view. In addition, a command line tool allows for direct control of access controls and other policies.

Chirp differs from traditional filesystems in that both client and server components are designed to be rapidly deployed without special privileges. Neither Parrot nor Chirp requires root access, kernel changes, kernel modules, or access to privileged ports. A user submitting jobs to any arbitrary grid may bring Parrot along with the job, and use it to securely access file systems established by the submitting user at another location. The user is in full control of the caching mechanism, caching policy, and failure semantics.

Chirp employs a custom protocol, which will be the subject of this paper. A client establishes a TCP connection to a server, negotiates an authentication method, authenticates, and then performs a series of remote procedure calls (RPCs) over that connection. If the connection is lost, a retry layer in `libchirp` is responsible for transparently re-connecting, re-authenticating, and re-opening files.

The base Chirp protocol is a set of RPCs that correspond very closely to the Unix I/O interface. Each of the calls in this is a "pure" RPC in the sense that the full arguments to the call are marshalled in `libchirp`, transmitted to the server, and the results are collected and verified before returning to the client. For example, here is a commonly-used subset:

```
open   (path,flags,mode) => fd
pread  (fd,length,offset) => (result,data)
pwrite (fd,data,length,offset) => result
close  (fd)
stat   (path) => metadata
unlink (path) => result
getdir (path) => dirlist
```

To this base set, we add two new streaming calls:

```
getfile (path) => (size,data)
putfile (path,size,data) => result
```

These two calls are "impure" RPCs, because they cannot be fully marshalled or unmarshalled without the help of the calling client. That is, the client presents the (perhaps GB of) file data in pieces as the TCP stream is ready to accept it. Thus, we describe Chirp as a *hybrid protocol* that has elements of both traditional RPC and streaming protocols. This allows Chirp to handle both large and small files efficiently. The challenge lies in designing the system such that the right operation gets invoked by ordinary tools.

III. Data Access Protocols Compared

Figure 2 compares several existing grid I/O protocols in detail, and explains why they have poor performance on small file workloads. In each case, the figure shows the network activity that occurs between a client and server upon an attempt to write one file, and then read another.

FTP [15] is a **streaming protocol** that has been extensively studied and optimized for grid computing [1], primarily because it is has a large existing user base. In FTP, a client initiates a *control connection* to a server, authenticates, and then issues commands to manipulate the filesystem. Simple commands, such as creating a directory or querying the size of a file are executed synchronously, and the result returned back to the user on the control connection. To send or receive a file, the client must request a *data connection* with either the `PORT` or `PASV` command, indicate the filename and direction of transfer with the `STOR` or `RECV` command, initiate or accept the new connection, and then transfer the necessary data. End-

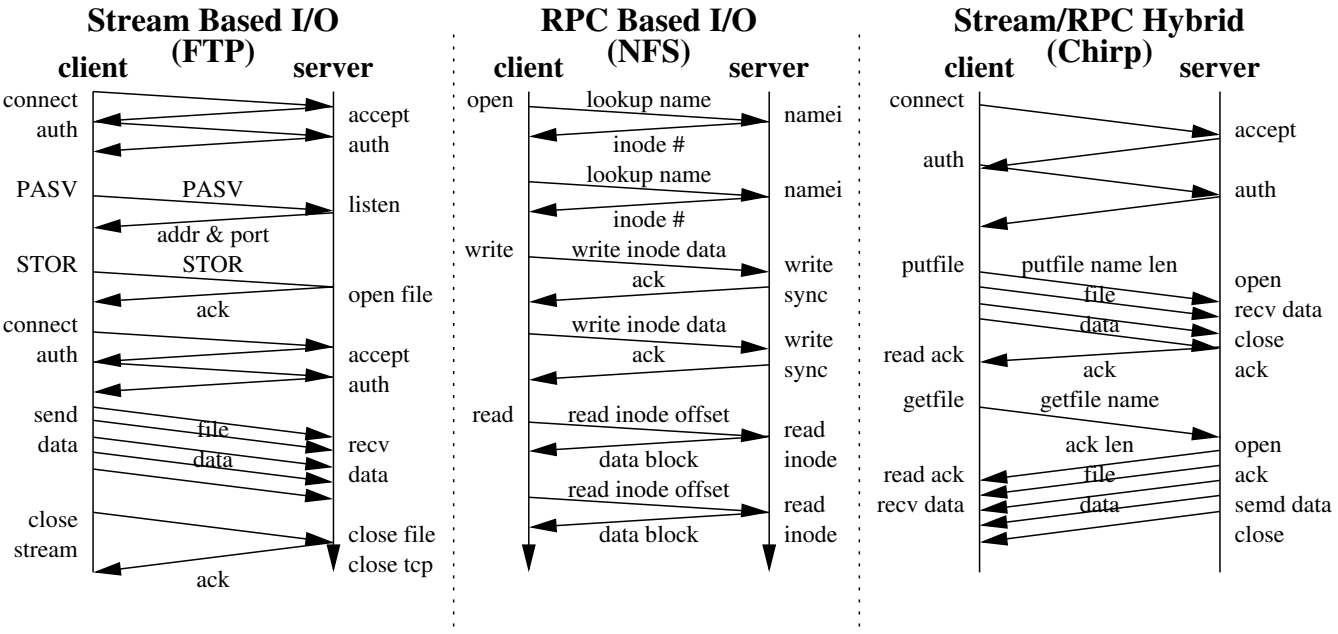

Fig. 2. Detailed Comparison of I/O Protocols

This figure compares three protocols for network I/O. On the left, FTP is shown as an example for stream-based I/O. Each file transfer requires a command on a control channel, followed by a new TCP connection to transfer data. In the middle, NFS is shown as an example of RPC-based I/O. Each I/O system call results in one or more synchronous network operations. On the right, Chirp is shown as an example of a hybrid protocol. Streaming of whole files may be interspersed with traditional RPCs.

of-file is indicated by the end of the TCP stream. To transfer more data, another connection must be made.

FTP and similar protocols that transmit one file per stream have both advantages and disadvantages. By using one file per stream, clients and servers are arguably simplified. Programs may use an FTP data connection as an input or output stream, with the same semantics as a local pipe. However, due to the vagaries of the system-call semantics, it is not always possible to distinguish between an ordinary end of file and a server crash or network failure. A higher-level check is needed to verify that a transfer is successful. The distinct data stream can be used to redirect data. For example, this is employed in third-party transfers between servers, and redirection to distributed storage devices in dCache [5].

However, from the perspective of moving small files, the separate connection and its attendant setup calls are disadvantages. Even assuming an existing control connection, each new transfer requires at least three network round trips (more for authentication) to transfer any file at all. Even for relatively large files, a new connection for each file causes TCP to restart its congestion control algorithm, resulting in less than optimal use of network bandwidth.

NFS [16] is an **RPC based protocol** widely used in clusters of all sizes, as well as a substrate for distributed and grid I/O services [25], [26], [27], [28], [29], [30]. NFS is designed for block-based access to files over a relatively low latency network. NFS is traditionally carried over UDP, each request and response contained in one packet. To access a file, a client must issue a series of `lookup` requests to map a path name to a file handle, which is usually the same as the inode number. The client may then issue small block read and write requests on that file handle. Block sizes are typically limited by implementations to be 1KB to 8KB. The protocol can also be carried over TCP, in which case block sizes may be larger.

The primary benefit of NFS is that it is very simple, which makes it amenable to kernel-level implementations, and accessible to modifications for research. In a local area network, small random accesses to files are very efficient, and result in a minimum of data transfer. Because the protocol is stateless, recovery from failure is very easy: unacknowledged operations are simply retried up to a timeout limit. However, this also places the obligation on the server to perform synchronous writes to disk before returning an acknowledgement.

NFS is not well suited to large wide area data transfers. A large file will be broken up into a large number of packets, each of which must be synchronously written and acknowledged. As network latency increases, data transfer speed drops precipitously. A less well-known limitation is this: the unlimited lifetime of file handles makes it impossible to have a robust user-level implementation of NFS. Kernel-level implementations traditionally use the inode number to construct unique file handles; this allows a client to return to a file at anytime without reprocessing the pathname. To provide the same service, a user-level server must remember on secondary storage all file handles ever issued.

Chirp is a **hybrid protocol** that combines elements of

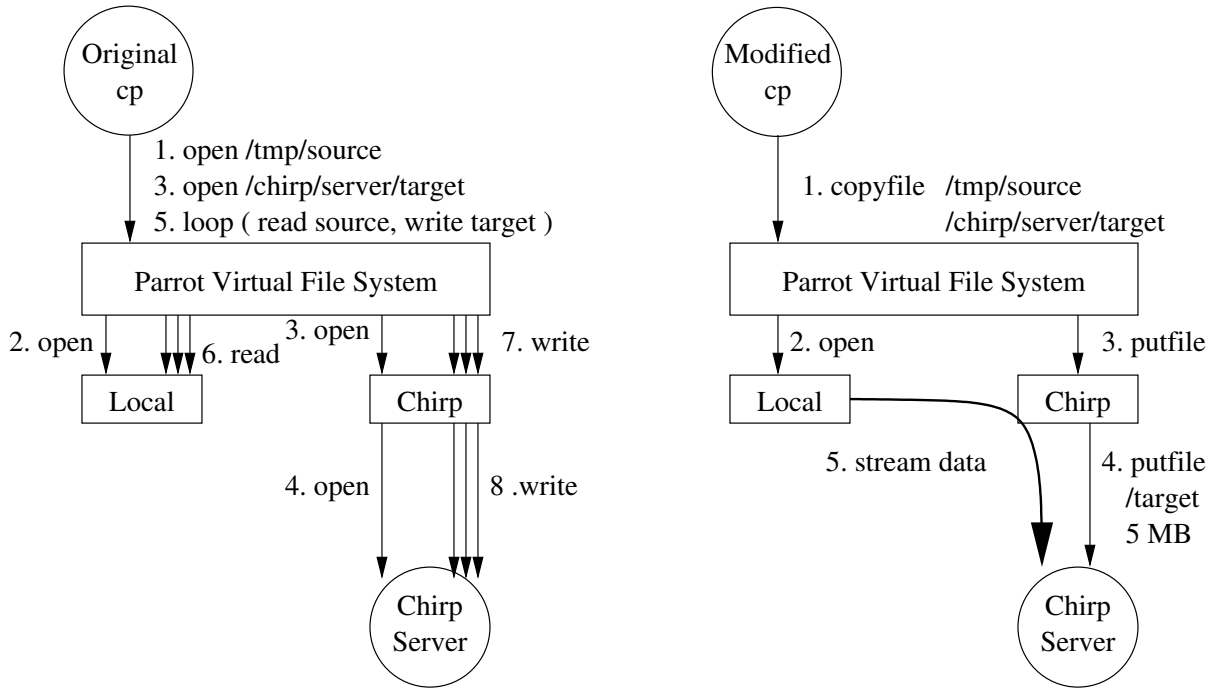

Fig. 3. System Support for Streaming I/O

Without high-level knowledge, it is difficult for the kernel to carry out an efficient file copy. A standard `cp` *tool opens the source and target files, and then issues small reads and writes to copy the data. However, if* `cp` *invokes a Parrot-specific system call* `copyfile`*, the more efficient streaming RPC can be used to transfer the data.*

both streaming and RPC protocols. A client makes a single TCP connection to a server, and then may make small RPCs, much like NFS over TCP. This also amortizes the cost of authentication, which may be quite high for systems such as Kerberos and GSI. When a large data transfer is needed, a header indicating the filename and transfer size is sent on the connection, followed by the data itself. This allows multiple transfers to occur in series, maintaining a large TCP window size on high bandwidth connections, and minimizing round trips on high latency connections. Unlike NFS, the use of direct file names allows for server-side processing, reducing round trips and allowing a simple user-level implementation.

We note that Chirp does not have the ability to stripe a file transfer across multiple TCP streams in the same manner as GridFTP [1], which is useful for rapidly achieving maximum bandwidth in a high latency, high bandwidth network. However, the need for such techniques is somewhat reduced by virtue of sustaining existing connections in the first place.

Based on this analysis, we propose that the hybrid Chirp protocol should have better performance on workloads that move large numbers of small files, because it minimizes the number of round trips necessary to deliver a file, compared to both NFS and FTP. However, it should also have large file streaming better or equal to FTP by virtue of re-using the existing stream across multiple files.

IV. System Support for Streaming I/O

Even if we provide a protocol that supports efficient small-file I/O, it can be difficult to match the protocol to the interfaces expected by existing programs. The hybrid protocol can easily be invoked through custom tools whose only job is to move data in and out of Chirp servers. However, users often do not care to learn new tools, and simply expect the existing interfaces to do a good job. We would like users to invoke the Chirp protocol through existing command line and graphical tools. Will they get good performance? The answer is *no*, and Figure 3 shows why. (We will show a solution to this problem below.)

Consider the user that invokes `cp` to copy a local file to a Chirp server. To do this, the user must be running the Parrot tool, which captures all of the system calls of unmodified applications, and re-interprets those that refer to remote filesystems. This is accomplished by associating user-level filesystem drivers with various parts of the namespace. For example, `open("/tmp/data")` is passed to the local filesystem driver to be executed without modification, while `open("/chirp/server/data")` is passed to the Chirp driver, which then invokes RPCs on the remote server.

The problem is that Parrot is not aware of the intent to copy a whole file. `cp` operates by opening the source file, then opening the target file, and issuing page-size reads and writes to copy the data. This situation is inefficient for several reasons. First, by carrying out the reads and writes one by one,

many round-trips will be made to the remote server. Second, all of the copied data is passed through the application, which has no need to actually examine it. Parrot cannot do anything more efficient, because it must support the full generality of the Unix interface: a program could do anything at all after calling `open`.

However, if the program makes Parrot aware of the intent to copy a file, something more efficient can be done. To accomplish this, we introduce a new system call called `copyfile` into Parrot. This system call simply takes two pathnames, a source and a target. We also create a new `cp` program that knows to invoke `copyfile`, rather than `open` followed by `read` and `write`. Note that we have not modified the host kernel nor the original `cp` program. Only Parrot and the modified program are aware of the system call. If run on the host kernel outside of Parrot, the `copyfile` call will fail, and the modified `cp` falls back to the old behavior.

Upon receiving a `copyfile` system call, Parrot internally opens the local source file, observes the file size, and then invokes `pufile` on the Chirp driver, which then streams the source data to the file server, using the hybrid RPC. This allows for the efficient movement of small files through the system. Although we have changed the system call interface and a standard tool, the changes are essentially invisible to the end user, who may use existing shells and scripts efficiently without modification. If other tools (such as graphical shells) are commonly used for copying files, then minor changes may be necessary to those tools as well.

The notion of introducing a higher-level interface to make system implementation more efficient is not new. The same concept is employed by object storage [31], [32], which proposes that disks should export an object-like interface, rather than individual disk blocks. This allows the underlying device to make more informed use of the available space. In the same spirit, we can imagine a number of operations (`copyfile`, `movefile`, `checksum`) that would be good candidates for elevation to system-call status, thus admitting a more efficient implementation.

Another idea similar to `copyfile` is `sendfile`, a Linux-specific system call that is designed to stream data from a file to a socket without passing through userspace. `sendfile` was designed specifically to support web and file servers. Why not overload `sendfile` to fit our needs? Unfortunately, `sendfile` accepts two file descriptors, and would require Parrot to translate and issue a remote `open` before discovering a file copy is in progress, and then issue a compensating `close`, resulting in two unnecessary round trips.

V. Performance of File Copies

To measure the performance impact of these protocol changes, we composed a series of experiments to compare the performance of each discipline – stream, RPC, and hybrid – in a common environment. Multi-protocol comparisons have some potential pitfalls. It is potentially misleading to compare three different protocols and implementations (e.g. an FTP server, an NFS server, and a Chirp server) side-by-side, because too many variables change: are the results an artifact of the protocol itself, server implementation, the workload generator, or tuning parameters?

Instead, we start by comparing the three styles of protocol using only Chirp. We construct a custom tool that performs a number of file copies from memory to `/dev/null` on the target machine. By avoiding an adapter (Parrot or FUSE) and the filesystem itself, we are exercising only the protocol variations, and not other aspects of the system.

Each measurement involves copying 1000 files ranging from 1KB to 4MB into a Chirp file server from a dedicated client and server on the same gigabit ethernet switch. To implement stream-per-file, we connect, authenticate, perform a `putfile`, retrieve the result, disconnect, and repeat. To implement RPC, we open the target file, issue a series of writes of a fixed blocksize, close the file, and repeat without disconnecting. To implement the hybrid protocol, we simply issue a series of `putfile` operations without disconnecting.

The latency of a network can have a significant effect on the performance of a protocol. The network hardware employed has a minimum latency of about 0.1 ms. To emulate higher latency networks, we implement an additional delay per operation on the target file server, and set it to 1ms to emulate a local area network, 10ms to emulate a wide area network, and 100ms to emulate an intercontinental network, which are not uncommon in wide-area grid computing systems.

The results of these experiments are shown in Figure 4. Network latency increases from top to bottom. On each graph, the size of files is shown on the X axis, and the time to transfer 1000 files of that size is given on the Y axis. Five lines indicate the transfer time for RPC (with three different block sizes), file-per-stream, and the hybrid protocol. Several items should be noted. As we might expect, the difference between protocols hardly matters in the 0.1ms network. With a low latency, the transfer time is dominated by the actual data to be transferred, even in the extreme case of 1KB files.

At the other end of the spectrum, the performance of the protocols differs by several orders of magnitude on the 100ms latency network. For very small files, the hybrid protocol only requires one round trip. RPC is three times slower than the hybrid protocol, because each file requires a network round trip to issue open, write, and close RPCs. Streaming is five times slower, because each file copy requires a round trip to open the connection, three to authenticate, and one to send the file and read back the result.

The RPC variants show significant slowdowns as the block size exceeds the file size, causing the performance to be dominated by round-trip delays. Streaming performance holds steady, until the total amount of data begins to dominate. Across the range, the hybrid protocol achieves better performance, and converges with streaming at very large data sizes.

At lower latencies, we see similar, if less pronounced behavior. Even on the 1ms network, there is a full order of magnitude difference in performance between the hybrid protocol and one file per stream.

It is interesting to consider the use of RPC protocols with

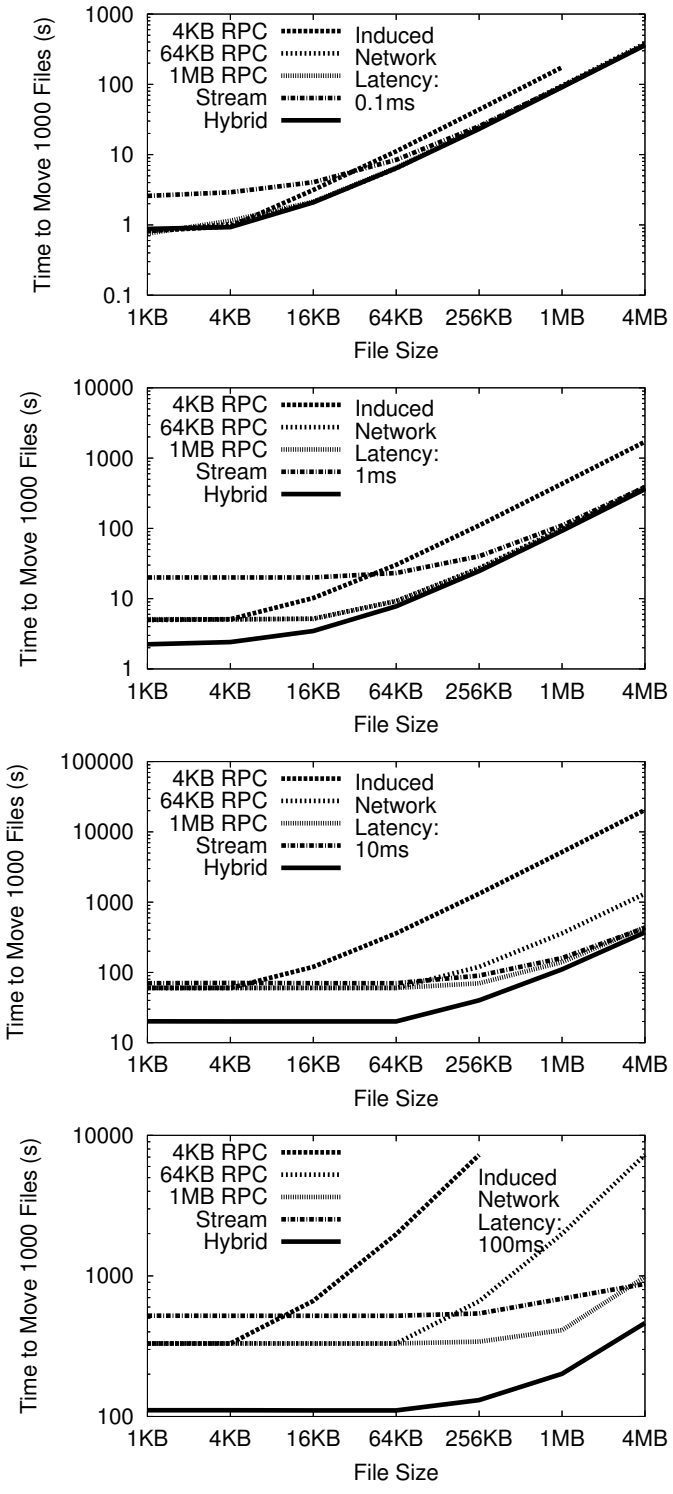

Fig. 4. Protocol Performance with Induced Network Latency

Each graph shows the time to copy 1000 files of varying sizes into a Chirp file server, using RPC, single file per stream, and the hybrid protocol. The induced network latency increases from top to bottom, representing a 0.1ms switch, a 1ms LAN, a 10ms WAN, and a 100ms VWAN. The hybrid protocol gives the best performance in all cases, with the widest margin on high latency networks.

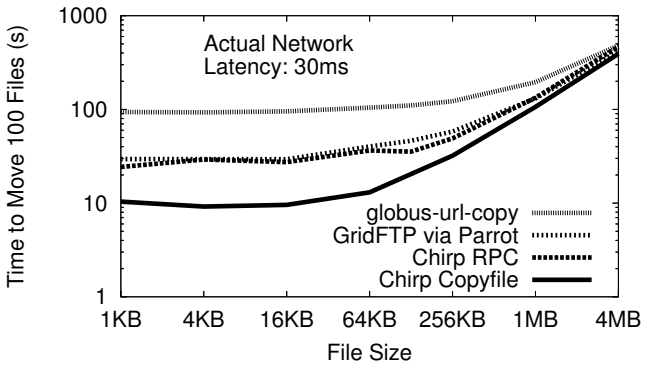

Fig. 5. Wide Area Protocol Performance.

The time to copy 100 files of varying sizes into a remote server over a real wide area network with 30ms latency. The hybrid Chirp protocol maintains the best performance by using a single connection and by minimizing round trips.

very large block sizes. A 1MB block size appears to give reasonable small file performance, and large file performance competetive with streaming: why not simply use an existing RPC protocol with a large block size? The problem with this approach is memory consumption: every open file would have a buffer of 1MB associated with it: an active system can easily have thousands of files open at once, and this does not scale.

Do these controlled results apply to real tools running on wide area networks? We answer this by running a similar measurement on a real wide area network measured latency of 30ms and maximum bandwidth of 2MB/s.

We compare the performance of several tools transferring 100 files of varying sizes over this network. (1) `globus-url-copy` is invoked once for each file to transfer data to a remote GridFTP server. This requires a control and data connection to be established for each file. (2) Parrot is used to copy files using the GridFTP protocol. The control connection is established once, and a new data connection is required for each file. (3) Parrot is used to copy files using the original `cp`, which invokes multiple Chirp RPCs for each file. (4) Parrot is used to copy files using the modified `cp`, which invokes `copyfile` once for each file. In all four cases, the data transfer is authenticated by GSI, but is not protected with encryption. Note that we cannot compare to NFS, as the protocol is blocked at campus firewalls.

The results are shown in Figure 5. `globus-url-copy` has a significant overhead for moving many small files. This can be improved by adding a component (Parrot) that can hold a connection open for an entire session. However, for moving many small files, the low-level optimization provided by Chirp `copyfile` is another factor of two improvement.

VI. Additional Small File Improvements

We can apply similar techniques to operations other than file copies to achieve performance benefits on small file workloads.

Third Party Transfer for Whole Directories. Both Chirp and FTP provide the ability to perform a third party transfer, in which a single node directs a file transfer between two other nodes, avoiding unnecessary network traffic. However, for moving a many small files, third party transfer of files is rarely worth the trouble: each transfer still requires some interaction with both the source and target server.

To make third party transfer more efficient for small files, we have modified the system to move entire directories on demand. When a client request a source to `thirdput` a directory to a target server, the server initiates a new connection with the client's delegated credentials, and then issues a series of `mkdir`, `putfile`, and `setacl` commands to reproduce the directory structure on the target server. This facility is invoked transparently by the `copyfile` or `rename` system calls, so that any attempt to `cp` or `mv` files between servers results in an efficient (and secure) third party transfer. Figure 6 shows the performance of directory copies using RPC and third party transfer on a file-by-file and whole directory basis.

Detailed Directory Listings. Listing directories is a very common (even impulsive) action performed by interactive users. A traditional `ls -l` on a filesystem invokes the `open`, `readdir`, and `close` system calls to fetch the directory names, and then an `lstat` on each file to retrieve its details. If each system call is mapped to its equivalent RPC in Chirp, then a large directory list will require many `lstat`s.

To optimize this situation, we introduce a new RPC `getlongdir` that fetches both the directory entries and the `lstat` data in a single round trip. To exploit this, an `open` on a directory invokes `getlongdir` and then caches the results for the successive `lstat` calls. Any other system call causes the cache to be discarded. This implements every `ls -l` invocation into a single round trip.

Recursive Deletion. Large deletions are also a common operation in temporary storage for a computing grid, where a workload may stage in data, work on it for a time, and then remove it. As with a long list, the system calls required for a delete involve listing each directory and then deleting each item individually. To address this problem, we introduce a new Chirp call `rmall` which deletes and entire directory on a remote server in a single round trip. System calls `unlink` and `rmdir` are mapped directly to `rmall`, so that existing programs such as `rm` invoke it transparently.

Active Storage Operations. Checksumming of files is also common in data intensive environments, often to provide an end-to-end check on data delivery. Using traditional RPC, a checksum requires the (potentially large) data to traverse the network to meet the (small) checksum program. A more efficient method is to move the executable to the data; this is known as *active storage* [33]. To this end, we provide several specialized active storage operations in the Chirp server. For example, a checksum on a remote file can be invoked with a single `md5sum` RPC, returning only the checksum result to the user. As before, this functionality is invoked by providing a modified `md5sum` program. Figure 7 shows the performance of a checksum using both RPC and active storage.

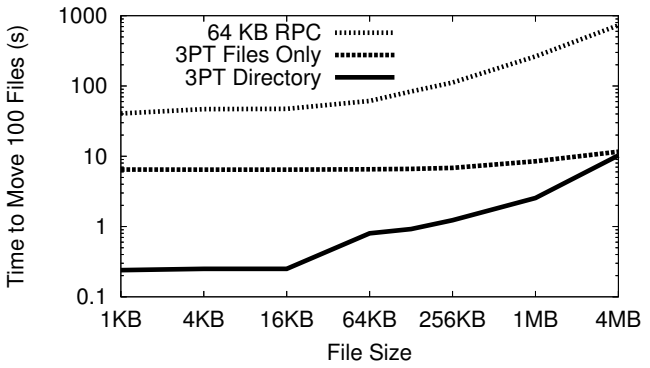

Fig. 6. Third Party Transfer Performance.
The time to copy a directory of 100 files from one server to another on the same 1Gb switch, directed by a client over a wide area network. Third party transfer (3PT) of an entire directory eliminates both wide area data movement as well as the overhead of setting up each individual file transfer.

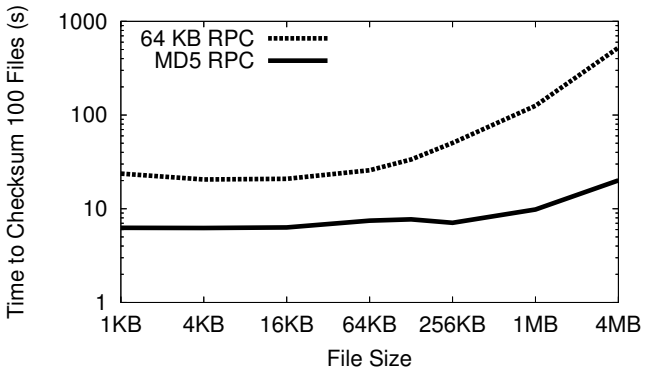

Fig. 7. Active Storage Performance
The time to checksum 100 files over a 30ms wide area networking using RPC and by invoking an active storage procedure. Active storage eliminates both data movement as well as multiple network round trips for each file.

Overall Performance Impact. The combined effect of each of these improvements on daily-use workloads is significant. To evaluate this impact, we return to the 30ms wide area network from before, and measure the performance of a script reflecting daily filesystem manipulations that might be performed by an ordinary user or by a job running in a grid computing system. The benchmark operates on a source tree package (Chirp version 2.3.1) consisting of 40 directories and 396 files. Most files are source modules of a few KB, but a handful of large data files bring the total amount of data to 5MB. This is the script:

```
cp -r  /tmp/pkg /chirp/a/pkg  # Copy In
mv     /chirp/a/pkg /chirp/b  # Move
ls -lR /chirp/b/pkg           # List
md5sum /chirp/b/pkg/*/*/*.c   # Checksum
cp -r  /chirp/b/pkg /tmp/pkg2 # Copy Out
rm -rf /chirp/b/pkg           # Delete
```

The two target file servers (`/chirp/a` and `/chirp/b`) are connected to the same low latency gigabit switch, and are separated from the client executing the script by the wide area network. We compare the performance of the original Chirp implementation to the improved Chirp protocol including all of the optimizations described in this paper. The script is repeated 10 times, measuring the wall clock time in seconds for each stage, yielding average and standard deviation:

Stage	Chirp Original	Chirp Improved
Copy Out	57.64 ± 0.14	29.90 ± 0.01
Move	166.08 ± 0.77	0.82 ± 0.12
List	17.83 ± 0.32	4.89 ± 0.37
Checksum	25.12 ± 0.06	3.37 ± 0.01
Copy In	104.97 ± 1.22	19.68 ± 0.72
Delete	27.10 ± 0.14	0.06 ± 0.01
Total	**398.78** ± 1.50	**58.73** ± 0.66

As can be seen, the overall impact of these techniques on ordinary scripts is significant. Operations requiring network traffic over the wide area are improved by a factor of two, while those that eliminate all but a single round trip (Checksum, Delete) improve by several orders of magnitude.

VII. Conclusion

Many workloads in a distributed system operate on large numbers of small files. These workloads are poorly supported by traditional protocols such as FTP and NFS, which were designed with the assumption of low latency networks. In this work, we have shown that attention to the low-level details of I/O protocols in a grid filesystem can yield significant performance benefits. In addition, we have demonstrated techniques for transparently introducing new bulk file operations into existing unmodified applications. With care, it is possible to build a system that performs well for both large and small files without requiring any change in user behavior.

Acknowledgements

This work was supported by National Science Foundation grants CCF-06-21434 and CNS-06-43229. We thank Todd Tannenbaum for loaning the use of a machine.

References

[1] 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*, 2000, pp. 161–163.

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

[3] M. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz, "Middleware for filtering very large scientific datasets on archival storage systems," in *IEEE Symposium on Mass Storage Systems*, 2000.

[4] S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, and S. Scott, "Freeloader:scavenging desktop storage resources for scientific data," in *International Conference for High Performance Computing and Communications (Supercomputing)*, Seattle, Washington, November 2005.

[5] M. Ernst, P. Fuhrmann, M. Gasthuber, T. Mkrtchyan, and C. Waldman, "dCache, a distributed storage data caching system," in *Proceedings of Computing in High Energy Physics*, Beijing, China, 2001.

[6] O. Barring, J. Baud, and J. Durand, "CASTOR project status," in *Proceedings of Computing in High Energy Physics*, Padua, Italy, 2000.

[7] D. A. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *ACM SIGMOD international conference on management of data*, June 1988, pp. 109–116.

[8] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *Annual Linux Showcase and Conference*, 2000.

[9] Cluster File Systems, "Lustre: A scalable, high performance file system," white paper, November 2002.

[10] E. Lee and C. Thekkath, "Petal: Distributed virtual disks," in *Architectural Support for Programming Languages and Operating Systems*, 1996.

[11] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The globus striped GridFTP framework and server," in *IEEE/ACM Supercomputing*, November 2005.

[12] T. Hacker, B. Noble, and B. Athey, "Improving throughput and maintaining fairness using parallel tcp," in *Proceedings of INFOCOM*, 2004.

[13] R. Brightwell and L. Fisk, "Scalable parallel application launch on cplant," in *IEEE/ACM Supercomputing*, 2001.

[14] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 3, no. 215, pp. 403–410, Oct 1990.

[15] J. Postel, "FTP: File transfer protocol specification," Internet Engineering Task Force Request for Comments (RFC) 765, June 1980.

[16] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the Sun network filesystem," in *USENIX Summer Technical Conference*, 1985, pp. 119–130.

[17] M. Silberstein, M. Factor, and D. Lorenz, "DYNAMO: DirectorY, Net Archiver and MOver," in *Third International Grid Computing Workshop*, 2002.

[18] J. Bresnahan, M. Link, R. Kettimuthu, D. Fraser, and I. Foster, "GridFTP Pipelining," in *Teragrid Conference*, 2007.

[19] D. Thain and M. Livny, "Parrot: Transparent user-level middleware for data-intensive computing," in *Proceedings of the Workshop on Adaptive Grid Middleware*, New Orleans, September 2003.

[20] D. Thain, S. Klous, J. Wozniak, P. Brenner, A. Striegel, and J. Izaguirre, "Separating abstractions from resources in a tactical storage system," in *IEEE/ACM Supercomputing*, November 2005.

[21] D. Thain, "Operating system support for space allocation in grid storage," in *IEEE Conference on Grid Computing*, September 2006.

[22] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, "A Security Architecture for Computational Grids," in *Proceedings of the 5th ACM Conference on Computer and Communications Security*, San Francisco, CA, November 1998, pp. 83–92.

[23] J. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," in *Proceedings of the USENIX Winter Technical Conference*, 1988, pp. 191–200.

[24] "Filesystem in user space," http://sourceforge.net/projects/fuse.

[25] D. Mazieres, "A toolkit for user-level file systems," in *USENIX Annual Technical Conference*, June 2001.

[26] D. Gifford, P. Jouvelot, M. Sheldon, and J. O'Toole, "Semantic file systems," in *ACM Symposium on Operating Systems Principles*, October 1991.

[27] A. Batsakis and R. Burns, "Cluster delegation: High-performance fault-tolerant data sharing in nfs," in *High Performance Distributed Computing*, 2004.

[28] D. Mazieres, M. Kaminsky, M. F. Kaashoek, and E. Witchel, "Separating key management from file system security," in *Symposium on Operating Systems Principles*, 1999, pp. 124–139.

[29] S. V. Anastasiadis, R. G. Wickremesinghe, and J. S. Chase, "Lerna: An active storage framework for flexible data access and management," in *High Performance Distributed Computing*, 2005.

[30] P. Honeyman, W. A. Adamson, and S. McKee, "GridNFS: Global storage for global collaboration," in *Local to Global Data Interoperability*, 2005.

[31] M. Mesnier, G. Ganger, and E. Riedel, "Object based storage," *IEEE Communications*, vol. 41, no. 8, August 2003.

[32] G. A. Gibson, D. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka, "File server scaling with network-attached secure disks," in *Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1997.

[33] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large scale data mining and multimedia," in *Very Large Databases (VLDB)*, 1998.