Transparent Remote File Access in the Minimum Intrusion Grid

Rasmus Andersen
University of Southern Denmark
anden@imada.sdu.dk

Brian Vinter
University of Southern Denmark
vinter@imada.sdu.dk

Abstract

This paper describes the implementation of a thin user-level layer to be installed on Grid resources. The layer fits in the Minimum intrusion Grid design by imposing as few requirements on the resource as possible and communicates with the server using only trusted and widely used protocols.

The model offers transparent, on-demand remote file access. By catching all application operations on input files, these operations are directed towards the remote copy on the server, thus eliminating the need for transferring the complete input file.

This implementation is targeted at the Minimum intrusion Grid project, which strives for minimum intrusion on the resource executing a job. In "minimum intrusion" lies, that a client need not install any dedicated Grid software, forcing the proposed model to use a user-level layer that automatically overrides GLIBC I/O calls.

Keywords: *Remote On-demand File Access, Transparency, Shared Library, Minimum intrusion Grid.*

1. Introduction

Currently, while still in its early stages, the Grid is primarily a domain for scientific applications. When such jobs are submitted to a commodity Grid, they are forwarded to a suitable resource that downloads all input files and executes the job.

However, due to the well known fact that often only fragments of input files are actually accessed and input files for this kind of applications can be exceedingly large, we would be better off using a resource client that downloads only the needed data from the input file.

In this paper, a layer providing on-demand remote file access is presented. This enables the resource to limit its file retrieval to a set of file-fragments that hold the needed data, rather than downloading the entire input file. A natural extension to this model is to apply prefetching to increase performance as well as encryption to ensure optimal security during the transfer on the Internet and the processing on the resource.

The model is targeted at the Minimum intrusion Grid, MiG, that provides a Grid infrastructure with minimal requirements on the resource, but it is portable to other Grid solutions.

1.1. Minimum Intrusion Grid

The philosophy behind the Minimum intrusion Grid, MiG, is to provide a Grid infrastructure that imposes as few requirements on users and resources as possible.

MiG is a stand-alone approach to Grid that does not depend on any existing systems, i.e. it is a completely new platform for Grid computing.

The challenge is to make the desire for minimum intrusion coexist with a large set of features, including scalability, autonomity (updating grid without causing users and resources inconvenience), anonymity (users and resources are anonymous to each other), fault tolerance, firewall compliance, etc.

In order for this client to comply to MiG, it must be ensured that it is entirely user-level and is installable without administrator privileges but still automatically overrides the native file system routines. Thus user applications need not be recompiled or rewritten using a custom MiG API.

1.1.1. File access in MiG One difficulty that users report when using Grid is file access, since files that are used by Grid jobs must be explicitly uploaded to a Grid storage element and result files must be downloaded equally explicitly. The MiG model introduces home-catalogs for all Grid users, and all file references are relative to this home-catalog. This eliminates all naming problems when implementing the proposed access layer, since MiG provides one simple access entry to a user's home-catalog.

1.2. Motivation

Job submissions in most Grids follow the intuitive flow: the user submits a job that Grid forwards to a resource that downloads the executable along with the input files, and finally the resource executes the job.

Often, only the first part or scattered fragments of input files are really needed. For Grid resources, this results in wasting storage and bandwidth by downloading entire files when only small pieces are needed.

Furthermore, lots of valuable time is lost as the job execution is delayed until everything has been downloaded. Of course, with huge input files the problem is more evident, and will be a limiting factor of performance, since the download time becomes a significant part of the total time from job submission to job termination. This suggests the need for a resource model which automatically downloads only the needed data.

This paper describes the problems we face when designing such a layer for MiG in section 2. The implementation of the corresponding solutions is given in section 3. Section 4 shows the performance of the client before we conclude in section 5.

1.3. Related Work

Numerous systems provide transparent access to remote files (NFS, AFS, etc.). Only a few run entirely in user space, for instance FUSE (Filesystem in User Space) and LUFS (Linux Userland FileSystem). LUFS enables mounting of a number of file systems including SshFS, thus allowing a user to mount files accessible by ssh in the file hierarchy. However, since these systems require root privileges to install a kernel module on the resource, they don't fit in the MiG model.

The Ufo Global File System [1] supports extending or altering the functionality of certain system calls by intercepting them in a user level module. The interception is achieved by standard tracing facilities. This strategy avoids the need for recompilation, relinking and administrator privileges. Using Ufo, one can transparently access personal accounts at remote sites using different protocols. The only drawback to this model is the interception method, which is quite expensive. Ufo is a great alternative for applications that issue a small number of system calls, but this is not the case for the Grid jobs in this project.

The ORFA client [3] provides efficient access to remote file systems using a preloaded light-weight shared library that overrides standard file access routines. The file management is handled by virtual file desriptors that enable remote files to be accessed and manipulated as local files. This is the exact same approach used in this project. However, no system providing on-demand file access nor a protocol supporting reading and writing ranges of bytes have been found.

2. Design

Providing on-demand transparent remote file access for a resource in a Grid environment first of all requires a protocol that supports retrieval of randomly requested data. Next, we have to make sure that certain file access routines on the resource are automatically overridden and redirected to the input file on the server. Finally, since the local file management is overruled, a local file management mechanism to ensure correct file access behaviour is needed.

2.1. File Data Transfer

HTTP/1.1 supports transferring ranges of bytes from a file in `GET` requests, but there is no range parameter in `PUT` requests, which is needed for remote writing. Since no file transfer protocol has been found to support range requests for both `PUT` and `GET`, a custom protocol is developed. The protocol should also support `OPEN` and `CLOSE` requests.

Due to the MiG restriction of firewall compliance, the connection is encapsulated in an SSH-tunnel. Thus the resource intrusion for data transfer consists of allowing SSH connections.

2.2. Catching Access to Input Files

The basic idea of the layer is to make user applications think the input files really exist in their entirety on the resource, yet they only exist on the server. This is achieved by interposing a user space file access layer between the application and the operating system. The purpose of this layer is to catch access to input files and direct them to the server holding the file.

Figure 1 shows the file access model. All file access routines issued from the user application are handled by the MiG file access layer that sends a request to the server (1). The server then replies (2) and an appropriate action is taken by the file access layer before the application receives the result (3). The dotted lines on the resource depict local management of remote files with and without server and kernel intervention, explained in 2.3.

Access to files not mentioned as input files are forwarded to the native file system (not shown in the figure).

Reading a block from an input file would result in a server request for the specified data that would be delivered in the buffer supplied by the application. The

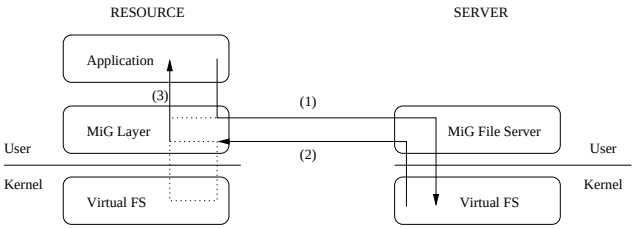

Figure 1. Overview of the MiG File Access Model

resource executing the job, would only have to download the executable before initiating the job. Subsequent file accesses result in fetching only the requested data and performing operations conforming to POSIX behavior on the fragment of the remote file.

2.3. File Handling

A shortcut to overriding the complete set of file manipulating routines is taken by actually creating the remote input file on the resource. Besides the obvious advantage of not implementing all file access routines, including maintenance of their evolution, we also avoid implementing complex UNIX functionalities. Instead, the input file is created on the resource and before any file access, it is ensured that accessed data is available.

Thus, as the dotted line on figure 1 illustrates, some remote file accesses, i.e. `open`, upon server response result in requesting the resource kernel system to create the file, whereas other calls, i.e. `read`, remain in the MiG layer on receipt of the server response and returns data directly to the application. Had the requested data already been fetched, it would have been returned immediately without server or kernel intervention.

Table 1 shows the set of file access routines that are being overridden.

Description	Functions
File Access	open, close, read, write (and similar stream functions)
File position	lseek, fseek, ftell, fgetpos, fsetpos, rewind
Synchronization	fsync, fdatasync, msync
Memory mapping	mmap, munmap, mremap
Memory protection	mprotect, mlock

Table 1. Overridden file access routines.

3. Implementation

The idea is to make the MiG file access layer do all file management, i.e. maintaining a file pointer and checking file access etc.

This is achieved by keeping a user-level structure, that upon an `open` call gathers all information about the file. The layer then uses the real `GLIBC open` call to create the file locally and maps that file into memory in its entirety. The file descriptor returned from the `open` call is finally given to the application. Creating and mapping the file in its entirety does not waste memory, since nothing is allocated until it is needed.

3.1. Virtual File Descriptors

All input files are described by virtual file descriptors. The virtual file descriptor is a structure containing several pieces of information about the file: the local file descriptor that is returned to the application, the socket descriptor, the length of the file, a pointer to the location in memory where the file resides, the filename, flags indicating access mode, etc.

Figure 2 shows how the layer interacts with the descriptor management in the kernel I/O subsytem. The figure also depicts the mapping of an input file in a snapshot where two fragments of the file have been accessed, see section 3.4.

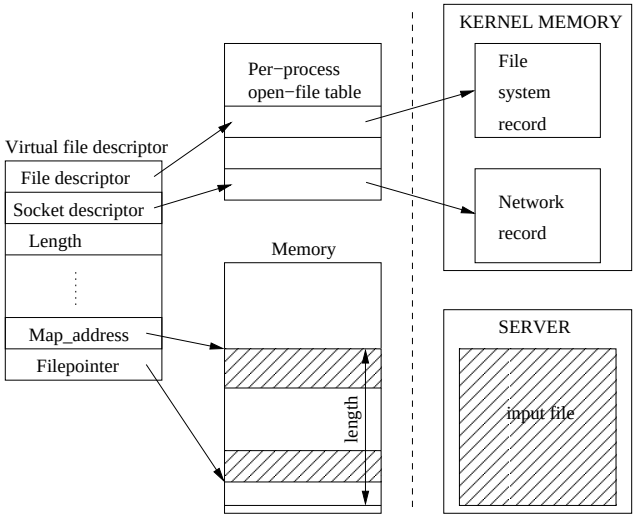

Figure 2. Interaction with the I/O subsystem.

3.2. File Transfer

One can never tell if user applications read reasonable amounts of data; one might read in chunks of 100 bytes, another in chunks of 1k. Clearly, sending a network request for such data amounts is very inefficient. For this reason, the layer always fetches blocks of multiples of 4k. This increases network performance and eases the file management.

Thus, if the user application requests bytes 1024 to 2047, the layer sends a request for the block surrounding the specified range, i.e. a request for bytes 0-4095, for a block size of 4k.

3.3. Catching File Access Routines

Linux supports preloading of user defined libraries to override a subset of the standard `GLIBC` functions. By setting the LD_PRELOAD environment variable the linker first checks the preloaded library for a matching symbol name. Thus, one can write a private set of `GLIBC` functions, while functions not overridden are automatically handled by `GLIBC`. This feature requires user applications to be dynamically linked, which most frequently is the case. Statically linked applications are not supported by this model.

If the application creates a file, that is not declared in the job script, the layer should forward the creation to `GLIBC`. Since the `open` call is overridden, the linker must be instructed to search for the next matching symbol. This is done using the *dlfcn* library that provides a handle, `RTLD_NEXT`, for finding the next occurrence of a function in the original search order after the current library.

3.4. Memory Mapping of Input Files

All remote input files opened by the user application are created on the resource and mapped into memory. Although the file is empty, it is mapped in its full length. Subsequent `read` calls will get data from the server and fill in requested pieces.

As shown in figure 2, the memory mapped file is highly fragmented, because only the needed data is retrieved, while only used space is actually allocated. Accesses to empty memory addresses within the scope of the file thus result in segmentation faults. Hence, a procedure to handle segmentation faults must be introduced. This is implemented using the `sigaction` system call, that invokes a procedure upon receipt of a `SIGSEGV` signal. This procedure gets the faulting address, determines the file owning the address, translates the address into a file offset, and sends a request to the server for the block surrounding the offset.

Mapping input files into memory has several advantages:

- The layer only deals with memory addresses when accessing files
- Direct memory access; the layer reads data from the socket directly into the correct location in memory. This prevents copying from a buffer.
- Reading prefetched or cached data is returned immediately without a system call.
- The memory mapped image may be returned directly to the application if it issues an `mmap` call.

3.5. File Access on the Resource

All functions that are overridden must exhibit the same behaviour as the corresponding `GLIBC` function and conform to ANSI C standard.

Figure 3 shows the data flow for a `read` call.

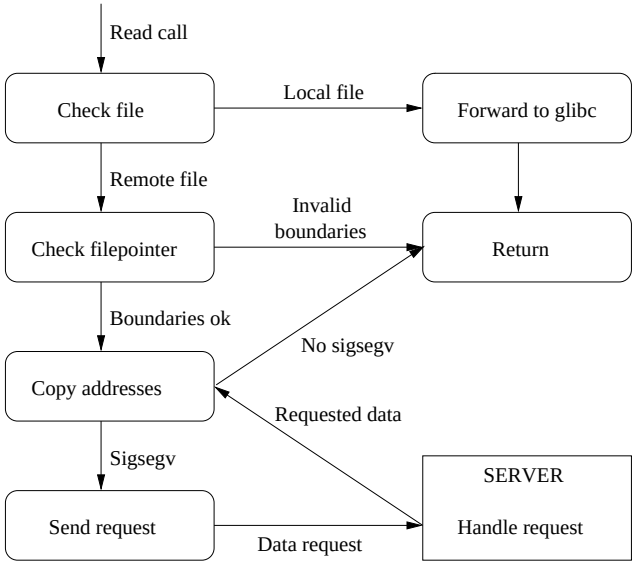

Figure 3. Data flow diagram for read calls.

First the layer determines whether the file is an input file or not. In the former case the call is forwarded to `GLIBC`, in the latter the virtual file descriptor is retrieved. Then the boundaries of the file are checked to avoid unnecessary network requests that do not get any data.

Next, the layer copies data directly from the file pointer to the user buffer. As explained in section 3.4, this may raise a `SIGSEGV` that invokes the procedure to get the requested data. If the application reads a chunk

larger than the network data-transfer size, the copying will continue raising `SIGSEGV`, which causes further data to be transferred. Eventually all data requested in the `read` call is fetched and the function can return. Hence the cycle in the figure.

Writing to a file results in sending the content of the supplied buffer to the server that writes to the real file. If the file is open for both reading and writing, the data is also copied to the location in memory pointed to by the file pointer. Seeking in a file moves the file pointer.

If the user calls `mmap` on an input file, and the access modes for the call match the open mode of the file, the layer just returns the address at which it mapped the file itself. Closing a file results in synchronizing the file to disk, unmapping it, calling the real `GLIBC close` on the file descriptor, and closing the socket.

3.6. Prefetching and Block Size

The choice of the block size depends only on the user application. If the application reads all blocks sequentially without much data processing, a large block size is preferred. The file access layer would then be able to continuously read the next block, thus always reading a block ahead and hiding the latency.

However, if the application only reads small scattered fragments of the file, reading ahead is useless and the application would have to wait for at large block to arrive. Thus, in this scenario a small block is preferred.

Since it is impossible to tell the access patterns of user applications in advance, we need to make the block size adapt dynamically to the application based on a 1-block read-ahead prefetching algorithm.

Every time the file access layer receives a block from the server, a thread is started to fetch the next block. The next time the application calls `read`, it is noted whether the prefetching is finished and whether the prefetched block is the one we need now. Based on these observations, the blocksize either increases, decreases or remains unchanged.

3.7. Security

In this project, only security issues regarding input files are considered. Executing a job on a foreign resource will always suffer from the inability to completely elude the resource administrator from surveying, copying or deleting input files. If the input files are highly confidential, we can never guarantee that the resource administrator cannot gain access to them. However, we can make it very hard to intrude on them.

First of all, everything is transferred on the Internet in SSH-tunnels. Secondly, in scenarios where optimal security is required, a user can choose to have the files block-encrypted on the file server. The file access layer locks the pages holding the block in memory and decrypts it at the moment before copying to the user buffer. After the copying, it is reencrypted and unlocked from memory.

This procedure hinders unencrypted data to be transferred onto a swap store medium. Of course, the data now lies unencrypted in the user buffer, but this is unavoidable.

4. Performance

4.1. Experiments

The model is tested, in 4 scenarios, in order to investigate the basic overhead, performance in an I/O limited application as well as an I/O balanced application, and finally in a scenario where only a small portion of a huge file is used.

4.1.1. Overhead In order to determine the basic overhead of the remote access protocol, the first experiment simply reads a file of size 1 byte and verifies the content. This experiment provides us a baseline for the performance of the remote access model.

4.1.2. I/O intensive application In the second test, the application checksums a 1 GB file. The calculations that are involved are so few and simple that the application is simply limited by I/O performance. Naturally the latency of getting a new page from disk is much less that retrieving the data through the remote access layer. Thus this is the kind of application that does not really perform well on Grid. Still, the copy-semantics should be faster than the remote access layer, since bulk transfer of a complete file is more efficient than the blocked access model.

4.1.3. I/O balanced application Here, a 1GB input file that requires some processing from the application is traversed. The input file contains a series of numbers that the application reads and computes a corresponding fibonacci value on one of the numbers. This test will show the benefit of using prefetching in combination with starting the job immediately without waiting for the input file to arrive and should prove favorable to the remote access model.

4.1.4. Partial file traversal Finally a large file containing a B+ tree of order 4 is searched using a random key. The test is run with 10 different keys. A B+ tree is an ideal structure for the remote access model since

it is designed to branch out in a large number of directions and to contain a lot of keys in each node. This ensures that the height of the tree is relatively small. Thus, only a small number of nodes, one in each level of the tree, must be read to retrieve an item.

4.2. Results

Table 2 shows the results of the 4 experiments. The results display the average of running the test 5 times.

The results of using the proposed layer (column 4) are compared to local execution (column 2) and to a model that just downloads the entire input file and executes the job (column 3). All experiments are performed on the same dedicated resource and server using a 100Mb network connection.

Experiment	Local	Copy	Remote
1 B file	0.0002	0.152	0.008
Checksum	50.11	130.10	114.03
Fibonacci	632.83	721.22	600.72
B+ tree	0.0002	30.692	0.0186

Table 2. Result of experiments (in seconds).

4.2.1. Overhead As the result of reading a 1 byte file reveals, using the proposed remote file access layer does not incur much overhead compared to the local execution. This experiment also shows that the layer is faster than using external *curl* to download the file.

4.2.2. I/O intensive application This result is surprising and is due to a combination of the prefetcher and the server. The prefetcher keeps increasing the block size, because it detects sequential access, and the python server is actually faster than the dedicated apache server that *curl* consults.

4.2.3. I/O balanced application The result of the Fibonacci experiment shows that the layer is able to hide the transfer time during data processing because it is faster than the download model. Amazingly, running this application using the proposed layer is faster than local execution. This is due to a combination of perfect prefetching and the design shown in figure 1. Almost all read requests are returned immediately by the file access layer without server or kernel intervention. Thus, all requested blocks are already in memory prior to the read calls. During local execution all blocks are fetched using a system call and expensive disk access.

4.2.4. Partial file traversal The B+ tree experiment performs excellent on the proposed model. The depth of the tree is 9, hence only 9 blocks are fetched plus an additional header block and wasted blocks from the prefetcher. Often the size of the B+ tree file is much larger than this one (357 MB), which in effect prohibits this kind of applications from execution with Grid implementations that use the download model. The speedup between the download model and the proposed model is a factor of 1650.

5. Conclusion

The MiG project aims at providing a new standalone Grid infrastructure that imposes as few requirements on users and resources as possible. The on-demand transparent remote file access layer is specifically designed towards this project without compromising neither the design requirements nor the features of this Grid project.

Using this user-level layer, it has been shown that a resource does not need to download entire input files. Instead it just starts the job and then downloads portions of the input files when needed. This functionality is achieved without requiring user application to use special API's or to be recompiled.

The results show that this model provides access to the Grid for a whole niche of applications that were previously impeded by unnecessary transfer of an enormous amount of data, namely applications using partial file traversal on huge input files, such as B+ trees.

This is also the case for applications such as high-energy physics applications that analyse relatively small fragments of massive physics database files.

References

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the operating system at the user level: the ufo global file system.

[2] S. F. Altschul, W. Gish, W. M. Miller, E. W., and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[3] L. P. Brice Goglin. Transparent remote file access through a shared library client, December 2003.

[4] I. Foster. A new infrastructure for 21st century science. *Physics Today*, 55(2):42–47, 2002.

[5] NDGF. Nordic data grid facility. http://www.ndgf.org.

[6] T. Tannenbaum and M. Litzkow. The condor distributed processing system. *Dr. Dobb's Journal*, 217:40–48, 1995.

[7] T. Ylonen. SSH - secure login connections over the internet. Proceedings of the 6th Security Symposium) (USENIX Association: Berkeley, CA):37, 1996.