Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

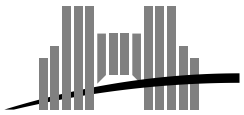
Transparent Remote File Access through a Shared Library Client

Brice Goglin, Loïc Prylli

December 2003

Research Report N° 2003-56

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr



Transparent Remote File Access through a Shared Library Client

Brice Goglin, Loïc Prylli

December 2003

Abstract

This paper presents the implementation of the ORFA client. ORFA aims at providing an efficient access to remote file systems through high-speed local networks such as MYRINET. The ORFA client is a lightweight shared library that may be pre-loaded to override standard file access routines to allow remote file access for any legacy application. In ORFA, virtual file descriptors have been designed to support POSIX behavior such as file sharing semantics so that remote files may be accessed and manipulated as local files. Local file access routines may still be used without any incompatibility with other libraries that modify their standard behavior. Finally, a network abstraction layer has been implemented in ORFA to efficiently use asynchronous interfaces such as GM without suffering from memory registration requirements.

Keywords: Remote File Access, Shared Library, Dynamic Linking, Transparency, Memory Registration, MYRINET, LINUX.

Résumé

Cet article présente l'implémentation du client ORFA. ORFA vise à fournir un accès efficace aux systèmes de fichiers distants à travers un réseau local hautes performances tel que MYRINET. Le client ORFA est une bibliothèque partagée très légère qui peut être préchargée pour surcharger les routines d'accès aux fichiers afin d'autoriser toute application classique à accéder aux fichiers distants. Des descripteurs de fichiers virtuels ont été mis en place pour supporter le comportement POSIX tel que la sémantique de partage de fichiers si bien que les fichiers distants peuvent être accédés et manipulés comme des fichiers locaux. Les routines d'accès aux fichiers locaux restent disponibles sans aucune incompabilité avec les autres librairies qui modifient leur comportement standard. Finalement, une couche d'abstraction réseau a été implémentée dans ORFA pour utiliser efficacement les interfaces asynchrones telles que GM sans souffrir des nécessités d'enregistrement mémoire.

Mots-clés: Accès aux fichiers distants, bibliothèque partagée, édition dynamique de liens, transparence, enregistrement mémoire, MYRINET, LINUX.

1 Introduction

The emergence of high performance local networks with zero-copy capabilities has given rise to applications based on direct data transfer between user applications and remote nodes. These technologies are now mainly used for communications between nodes running massive parallel applications on clusters.

Several mature projects such as GPFS [SH02], LUSTRE [Clu02] and PVFS [CLRT00] have already been proposed to provide efficient file distribution in cluster environment. These systems generally focus on data striping and work parallelizing across several servers to get scalable models that can fit the needs of parallel applications. We developed ORFA (*Optimized Remote File-system Access*) to study the impact on remote file access of the efficient use of the underlying material, especially the network subsystem. Indeed, high performance networks such as MYRINET [BCF+95] may be as useful for remote file accesses as for communications between nodes of a parallel application.

In this paper, we present the implementation of the ORFA client as a shared library which provides a fully transparent support for remote file access for any legacy application without rewriting or recompilation. We first present the context, related works and ORFA overview in Section 2. ORFA remote file access implementation is detailed in Section 3. Its inclusion in a fully transparent shared library is described in Section 4. Before concluding, we present network and memory registration issues in Section 5.

2 Context and Related Works

High performance computing requires both communication between nodes and access to data storage to be as fast as possible. Cluster architectures are based on high performance networks such as MYRINET or INFINIBAND [Fut01]. They all provide their dedicated user-level API such as GM [Myr03]. This has led to very efficient implementations of MPI layers for communications between nodes running parallel applications.

On the other hand, data access in such environments cannot immediately benefit from these interfaces because they have not been designed for such an usage. Several projects such as PVFS or GPFS have already been proposed to provide high performance data access through MPI-IO implementations. However, they generally focus on the parallelization of work across several servers and striping of data across multiple storage systems to provide scalable models.

2.1 Distributing Files

Figure 1 describes the UNIX file system stack and shows what entry points are available to developers who wants to implement a new file system. When an application accesses a file, its call is converted into a system call by the standard library (usually the GLIBC). The kernel system call handler passes the request to its VFS layer (*Virtual File System*) where the target path or file descriptor is translated into an inode or a file. The VFS uniformizes the interfaces of all supported file systems and transmits the user requests to specific routines of the file system that contains the target. Data accesses are buffered in the *Page Cache*. Real disk accesses are handled by the *Block Device Layer* which uniformizes the different physical partition types.

Implementing a distributed file system may first be done through the block device layer. A network client may be added to usual IDE or SCSI driver. This Network Block Device model (see Figure 2(a)) aims at mounting a remote partition.

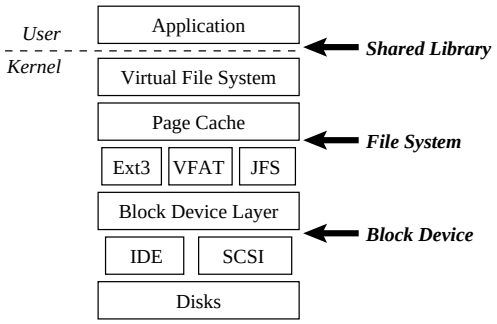
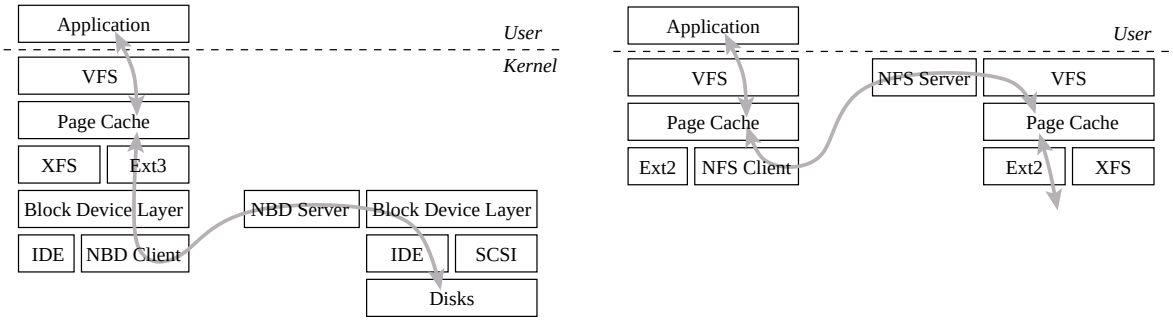


Figure 1: Schematic View of the UNIX File System Implementation.



(a) Network Block Device model.

(b) Network File System model.

Figure 2: Overview of Network Block Device and Network File System Models.

The most commonly used model is described on Figure 2(b). Its first implementation, NFS (*Network File System*), has led to lots of variants such as CODA and INTERMEZZO which are based on complex caching protocols and the ability for the client to work even if the server is down. However, several performance bottlenecks in these models prevent from being used in a cluster environment. Several parallel systems such as PVFS, GPFS and LUSTRE were also developed to fit cluster needs, that are high performance, scalability and enforced consistency.

The easiest way to access remote storage is a user-level remote file-system client. This model allows to bypass the kernel. This allows to directly connect the user application buffers to the remote storage through zero-copy network transfers. On the other hand, it removes the VFS inode, directory entry and page caches. This could be an advantage for developers who want to avoid any consistency trouble or use their own customized cache.

2.2 Related User-level Implementations

Several projects already proposed a user-level client to access a remote file system. This method has the advantage of being very simple to use since no administrative rights are required contrary to kernel module implementations. Moreover, development and debugging are much more easier.

For instance, GNOME VFS was developed to provide a general purpose file access for GNOME applications, even for files that are stored on HTTP server or in a compressed archive. However, legacy applications cannot use GNOME VFS since they have to be recompiled to fit the new API.

The SAMBA system also provides a specific API but adds the possibility to transparently use it in common applications. This was done through the SMBSH program which launches a new shell that looks for a SAMBA-like path in user accesses and redirects them to the SAMBA server. This method seems to be very interesting but does not support many applications (see Section 4.1).

These projects aim at general purpose access and are thus not designed for high performance data transfer such as in a cluster environment. PVFS (*Parallel Virtual File System*) is one of the most famous parallel file systems. It is based on the distribution of the workload and data striping across several servers. Its first client implementation exported a specific API which also required to recompile applications. The need to make PVFS available for any application led to use the `LD_PRELOAD` environment variable to allow a library to replace GLIBC symbols by PVFS symbols (see Section 4.1). However, the client was finally ported into the LINUX kernel to benefit from the automatic support for all UNIX functionalities (see Section 3.2) and VFS caches. Most file systems have now been ported into the kernel for similar reasons. However, this presents the disadvantage of imposing to use VFS caches and fitting its constraints.

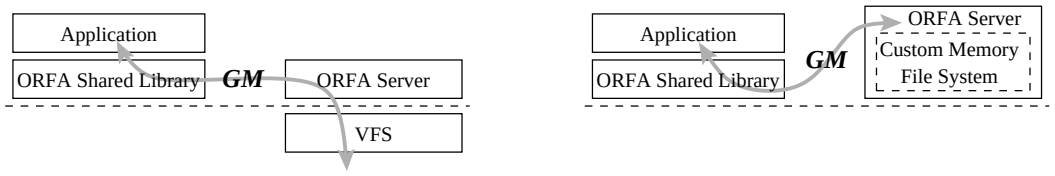
The DAFS consortium (*Direct Access File System*) designed a new protocol to benefit from the emerging RDDP techniques (*Remote Direct Data Placement*) such as RDMA (*Remote Direct Memory Access*). This allows to directly connect user buffers on the client's side to the remote storage system. A DAFS implementation was proposed in [MAF+02]. An optimistic model with client initiated data transfers is described in [MAFS03]. The DAFS client exports a highly specific API that was designed to make the most out of the network architecture and thus requires an important rewriting of application.

2.3 ORFA Design

ORFA (*Optimized Remote File-system Access*) was designed to study the impact of an efficient use of the underlying network when accessing remote files in a cluster environment. The aim is to use

the fastest link between user application buffers on the client's side and files that are stored on the remote server.

The first ORFA implementation is described in [GP03a]. Development has been done on 32 bits INTEL architecture workstations running LINUX kernel 2.4.20 and a standard GLIBC. We chose to avoid any overhead in the path from buffers to storage by first drastically reducing the client intelligence. This was achieved by avoiding any cache on the client side and designing the ORFA protocol around the usual POSIX file access API (`open`, `read`, etc.).



(a) Basic ORFA model.

(b) Custom memory file system.

Figure 3: Overview of ORFA Model.

ORFA model is summarized on Figure 3. The ORFA client is based on lightweight user-level client that intercepts I/O calls from the user application and transmits them to the remote server. The remote server may be a user program converting client requests into local file access (see Figure 3(a)) or accessing a custom file system implemented in user-memory (Figure 3(b)). Communications between these two instances were made efficient through the use of a native MYRINET network API (especially GM). Performance evaluation of this implementation is provided in [GP03b]. The memory file system was implemented to avoid file system and disk bottlenecks when evaluating the ORFA protocol.

ORFA goals are similar to DAFS while its implementation is totally different. ORFA client is totally transparent and may be used by any legacy application. It provides a remote file access library which is made transparent by a dedicated wrapper.

3 Remote File Access in ORFA

The ORFA client role is to convert applications I/O calls into requests sent to the remote server through the network as fast as possible. We have designed a custom protocol to transmit these requests and we have implemented a dedicated interface to provide this conversion.

3.1 Application Programming Interface

The design of the new API was guided by the need to get a transparent shared library. On the other hand, several other projects such as DAFS [MAF+02] explicitly chose to use a totally different API so that any application has to be rewritten and recompiled to use DAFS. This constraining design choice was necessary for the custom API was designed to make the most out of the underlying network and I/O architecture. The DAFS client provides a fully asynchronous I/O interface with completion groups, allowing rewritten applications to efficiently overlap computation and I/O. It also avoids several troubles with memory registration mechanisms (see Section 5) because the

developer is forced to use some DAFS specific memory management or other non-file access specific routines.

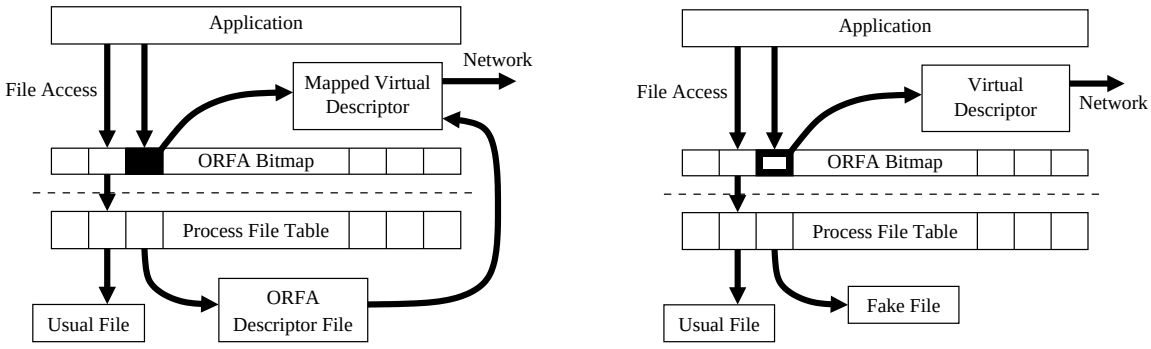
Our approach differs from this one since we want our client to be usable by any legacy application. Our remote file access API was thus designed around the usual POSIX API (`open`, `read`, etc.). Therefore, the ORFA client contains a remote access library with a POSIX-like API. Section 4 describes how these routines are made transparently accessible by any legacy application.

3.2 File Descriptor Semantics and Unix Functionalities Support

The ORFA client interface is based on descriptors for any open file or directory. These descriptors are given to the application which manipulates them as standard descriptors for local files or directories. However, ORFA descriptors contain much more information since they are used to retrieve the location of the real file on a remote server.

Standard descriptors are integer values that can be dereferenced into a kernel data structure through the process file table[1]. Due to ORFA user-level implementation, the behavior of ORFA virtual descriptors has to be simulated through user-level structures. This is problematic since the kernel manages the behavior of file descriptors under several complex UNIX functionalities such as `fork` or `exec`.

This behavior is often known as *file sharing semantics* and consists of the ability to share an open file descriptor between two entries in a file table or even between several processes. While the first case is easy to support, sharing a virtual descriptor between two processes may be difficult since their address spaces are different. Shared memory was thus required and achieved by using memory mapped files. Actually, the real file descriptor that is used to access this real mapped file is given to the application as a faked descriptor. This provides an automatic support for `dup` and `fork`. Finally, each local or ORFA open file corresponds to a descriptor in the process file table. The ORFA library maintains a bitmap describing whether they are ORFA or usual files (see Figure 4(a)).



(a) Memory mapped descriptor.

(b) Descriptors are initially not mapped.

Figure 4: ORFA Virtual File Descriptor Model.

We also enhanced this implementation to support the `exec` system calls and the *close-on-exec* flag. Memory mapped virtual descriptors are kept during `exec` in the same way a usual file is.

[1]Each process keeps in its kernel space a table of pointer to `struct file*` structures describing each open file.

However, the newly created process cannot access the ORFA bitmap describing ORFA open files since the whole address space has been deleted. The bitmap is thus passed during `exec` through an environment variable.

However, the high overhead of memory mapping compared to open and close operations[2] lead us to delay memory mapping until really required. Each ORFA virtual descriptor is kept in usual memory until a `fork` or a `exec` occurs. It is still necessary to give a file descriptor index to the application that will not change when the memory mapping really occurs. An index in the file table is thus reserved by duplicating the standard error file descriptor (see Figure 4(b)). When a `fork` occurs, it is closed and replaced by the newly memory mapped file descriptor.

Thus, the ORFA API provides routines to be called when any of these special UNIX functionalities has to be used. This provides the ability to access ORFA remote files in almost any situation. Memory mapping of remote files remains impossible since this feature explicitly requires to deal with the page cache and thus needs to be implemented in the kernel. We could have imagine to map a local copy of the remote file. However, this idea is only suitable for read-only access since the user-level library cannot precisely and cleverly trace file modifications in this mapping.

4 Local Access Transparency in ORFA

As describe in the previous section, the ORFA remote access library requires the developer to rewrite and recompile its application to fit the ORFA API and use the specific `fork` or `exec` routines.

We may have continue modifying this API to fit the needs of applications running on clusters, that is for example overlapping file access with computation through an asynchronous interface. This is what DAFS was designed to do. But, this prevents from being transparently usable by any legacy application without a strong rewriting effort and recompilation. Moreover, it still requires to access usual files through the standard POSIX API.

We chose to focus on transparency, so that any application can use our implementation. We thus have to keep a POSIX-like API. The next section describes how it is made transparent.

4.1 Overriding POSIX API

Most existing applications are dynamically linked. The compiler processes the application code and checks that non-defined symbols exist in available shared libraries. The final binary file does not include these libraries (except when a static linking is explicitly asked). This has the advantage of reducing occupied disk space by factorizing library binaries, avoiding application recompilation when a new library is released and providing a more flexible environment to the user who can choose which library is loaded and how.

Most modern UNIX systems provides several environment variables to modify the way shared libraries are dynamically loaded at runtime. The `LD_PRELOAD` variable allows to define a shared library that the system must load before any other library. As linking is made using first found symbols in the process address space, any symbol defined in a pre-loaded library will be used instead standard library symbols.

The ORFA transparent implementation is thus based on a pre-loaded library that redefines any I/O symbols of the GLIBC. Any application I/O call is then intercepted by a wrapper rou-

[2] `open` and `close` are almost only system calls (about 500 ns on a modern system). On the other hand, `mmap` requires to flush the Translation Look-aside Buffer (about 200 μs).

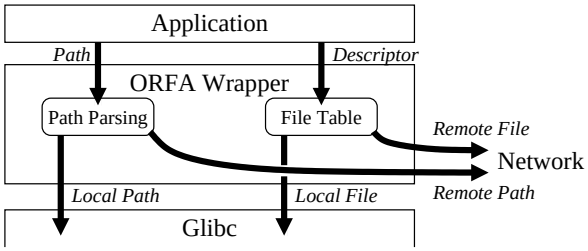


Figure 5: Schematic View of Local and Remote Access Handling by the Wrapper in the ORFA Client.

tine in the ORFA client. Call parameters are then parsed to check whether the target file is remote or local (see Figure 5). When accessing files through their pathname, a faked mount-point `/REMOTE@host/` has to be used to target files that are available on server *host*. When using descriptors, the ORFA bitmap is used as described in Section 3.2.

Remote calls are then handled through the ORFA remote file access library that we described in Section 3. Local calls are given back to the standard libraries. Moreover, the standard routines are necessary to manage virtual descriptor that are memory mapped (see Section 3.2). But, the standard library symbols are hidden by ORFA symbols, what makes them unavailable for normal dynamic linking.

Several projects such as SMBSH used a copy of GLIBC code for this purpose. This method may be satisfying for basic I/O routines that are implemented as system call wrappers. However, most other routines are much more complex. This could be an issue since GLIBC evolution could change this implementation. Moreover, portability requires to copy each architecture and operating system specific code.

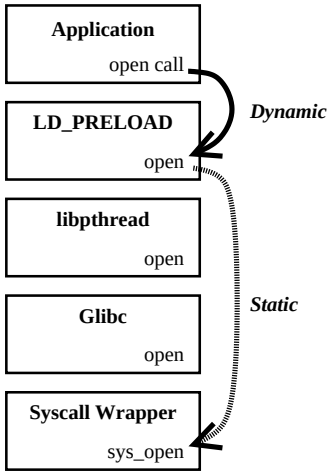
4.2 Transparency among other Libraries

The best way to ensure that our implementation is fully transparent is to call GLIBC routines as the application would have called it if the ORFA library were not present. The ORFA shared library is able to find the original GLIBC symbol that it has overridden. This is done through the `dlfcn` library which aims at programming dynamic loading. In addition to its interface to open any shared library at run-time, `dlfcn` provides the special `RTLD_NEXT` handle to find the next occurrence of a symbol in the search order after the current library. This was explicitly designed to implement a wrapper around a function in another shared library. That is exactly what ORFA does with GLIBC I/O routines.

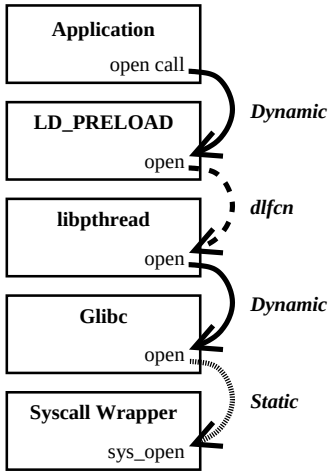
While making our implementation multithread-safe did not show any issue, this raised the problem of ORFA interaction with other libraries. Lots of shared libraries modify some standard routines by redefining their symbols. For instance, `libpthread` fixes the behavior of several system calls under cancellation by another thread. Programs that are linked with this library at compile time will load it over the GLIBC at run-time so that some GLIBC symbols are overridden.

Figure 6(a) describes a basic implementation such as SMBSH which cannot handle `libpthread` presence and thus does not use the modified `open` symbol. On the other hand, the usage of `dlfcn` in our implementation ensures that our shared library will not break other library impact (see Figure 6(b)).

Actually, standard libraries such as GLIBC use several different symbols for almost the same



(a) Copying the GLIBC code to simulate normal behavior may not be compatible with other libraries.



(b) `dlfcn` library usage keeps normal behavior.

Figure 6: Interaction of a Pre-loaded Library with other Shared Libraries when handling Local File Access.

purpose. For instance, opening a file may generally be done through `open` which calls `__open` which is based on `__libc_open`. The `libpthread` library then only redefines `open` so that the other symbols are still available to a developer needing to bypass `libpthread` changes. We thus have to independently wrap each of these symbols.

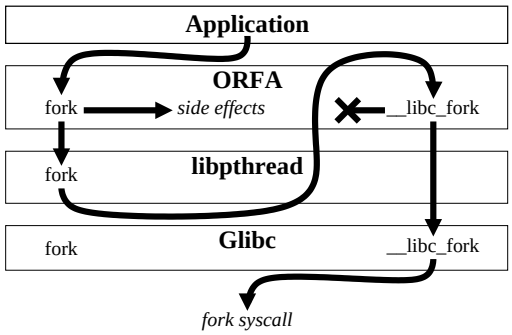


Figure 7: An application calling `fork` goes two times through the ORFA library since the `libpthread` implementation uses the GLIBC `__libc_fork`. ORFA side effects are only applied in the first nested call.

Such a nested GLIBC implementation implies that the application may actually use several ORFA wrappers for one I/O call from the application. We thus had to ensure that the side-effects of ORFA routines are not duplicated. This is especially important for special UNIX functionalities such as `fork` and `exec`. As describe at the end of Section 3.2, the ORFA API provides several special routines to emulate file descriptor behavior. The ORFA wrapper thus has to intercept `fork` and `exec` to use these routines. Figure 7 describes the the way a `fork` call is handled

through the ORFA shared library, its unique side-effects and GLIBC nested implementation.

The stream I/O interface (`FILE*`, `fopen`, ...) is also supported by ORFA. Instead of simulating virtual streams, we intercept several low-level routines that use the standard file descriptor interface. The ORFA library implementation is summarized by Figure 8.

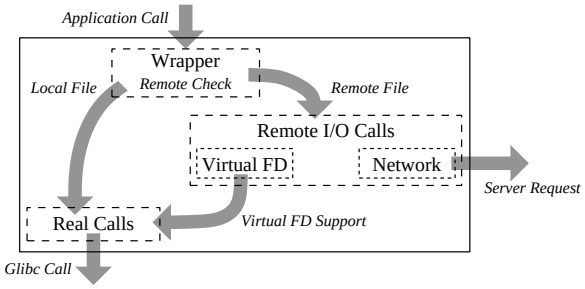


Figure 8: ORFA Shared Library Client Internals.

4.3 Interception Overhead

Access type	Base Cost	ORFA Overhead
Descriptor (`fstat`)	576	39
Path (`stat`)	1460 +7×*pathlen*	285 +10×*pathlen*

Table 1: Overhead in nanoseconds of I/O Calls Interception in ORFA Shared Library. Measurements were done on a dual Xeon 2.6 Ghz running LINUX 2.4.20.

We measured the overhead of the ORFA wrapper on local access. Results are summarized by Table 1.

Interception of application calls by the ORFA wrapper has an almost negligible cost, except when the target file is defined through a very long path. The reason is that the ORFA client has to parse this entire path to check whether it concerns a remote server or not. The slope of this overhead lead us to the conclusion that our parsing algorithm remains under-optimized compared to the VFS path lookup. Nevertheless, files are generally accessed through file descriptors.

The overhead we have measured only concerns the whole check for remote files. Remote file access also requires network communications. However, MYRINET interfaces may totally overlap communications with computations in the node. This fits our goal of implementing the ORFA client as a lightweight shared library.

5 Network Usage and Memory Registration

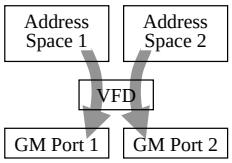
ORFA was designed to efficiently use high bandwidth local networks in order to get the fastest way for data to go from client application buffers to remote storage system. The available implementation may run on MYRINET networks through their GM (or BIP [PT97]) interface.

The transparent access layer that we described in previous section has to fit the requirements of these asynchronous network interfaces. This required to implement the ORFA network abstraction layer without useless memory copies or any mechanisms that could have broken the efficiency of network communications.

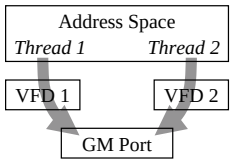
5.1 Network Usage in ORFA

Each time the ORFA remote interface has to access a remote server, it asks its network abstraction layer. Network communication may concern any node on the network. However, accessing an open remote file or walking in a directory often implies the same server during several consecutive accesses. Moreover, closing unused connections would have add to much overhead. Connections are thus kept open even if they are not currently used. This was done in a primary ORFA implementation over TCP. However, the GM MYRINET interface does not use connections. A single GM *Port* has to be open and provides point-to-point communications. This port is kept open as long as possible.

The ORFA client does not provide a mount point as a file system implemented in the kernel does. The `/REMOTE@host/` path is a totally faked mount point. However, it could have led to the idea of sharing connections to a server between several processes. Such a TCP implementation requires multiplexing of input and output in a socket, and thus synchronization techniques. This would only have reduced the number of open sockets on both sides but would not have enhanced client performance.



(a) Different address spaces use different GM ports even for the same descriptor.



(b) Threads of the same address space share a GM port even for different descriptors.

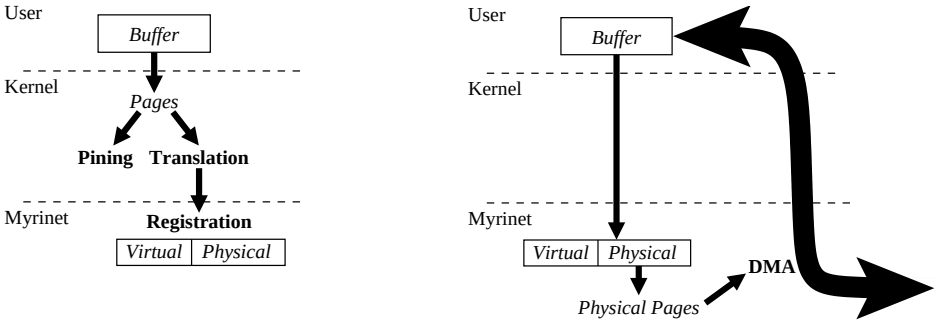
Figure 9: GM Port Usage through different Address Spaces, Threads or Virtual Descriptors.

On the other hand, our GM implementation is constrained by the fact that a GM port cannot be shared across different address spaces. Thus, a process and its `fork` child sharing a file descriptor cannot access it through the same port (see Figure 9(a)). We thus may keep a single GM port in each process, close it during `fork` (and `exec`), and reopen it when a network access occurs.

Nevertheless, multithreaded applications use the same port for all threads of the same address space (see Figure 9(b)). Several synchronization primitives are required to ensure communication multiplexing.

5.2 Direct Memory Access and Memory Registration

Using an asynchronous network interface such as GM requires to deal with an unusual communication model. The user program has to pass a send or receive request to the operating system (or to the network interface with MYRINET) and wait for the completion event. Data are transfered between the host memory and the network through a DMA. High performance network



(a) Memory registration.

(b) Sending.

Figure 10: GM Memory Registration Model. User buffers are first pinned in physical memory and their address translation are registered in the NIC. All following communications will be directly processed by the NIC which can initiate DMA transfer through its memory and the user application.

driver such as GM, BIP, QSNET [PcFH+02] or INFINIBAND are based on virtual memory user requests that are translated into physical addresses by the network interface. This translation may be done through a replication of the host translation table in the NIC. QSNET uses this method on QUADRICS networks. This implies to modify the operating system and requires a huge memory amount in the NIC.

On the other hand MYRINET interfaces only keep a translation cache that may be out-of-date. It is then required that a memory buffer is not swapped out or swapped into a different physical area between the request submission in the host and its processing in the NIC. This is the reason why GM, BIP and more general API such as VI [SASB99] provide a memory registration model. A buffer that has to be used in a communication is registered to the NIC and the operating system pins it in physical memory to prevent swapping out. This model is summarized on Figure 10. It allows direct data transfer between a user-level buffer in the application and the network without any memory copy.

The issue is that memory deregistration requires to interrupt the NIC processor (the LANAI) to invalidate its translation cache. This is the reason why deregistration should be used in a carefully to avoid its important overhead.[3]

Several interfaces such as BIP provide an automatic registration model making it transparent to the user. On the other hand, GM, VI and INFINIBAND ask the user to register its communication buffer by himself. This allows him to optimize its memory organization and registration usage to reduce deregistration impact.

5.3 Memory Registration in ORFA

ORFA was mainly implemented on GM and thus has to deal with memory registration. Since legacy applications were not initially compiled for such an usage, their I/O buffers are not managed in order to efficiently deal with memory registration.

[3]Memory deregistration lasts a few milliseconds while the network latency is a few microseconds.

DAFS faced the same problem since their implementation is based on VI or GM software layer. However, this issue was solved by adding corresponding routines in the DAFS API. As the developer already has to rewrite its application to fit with the custom asynchronous I/O API provided by DAFS, it is also required to register potential I/O buffers to DAFS. Moreover, several registered memory allocation routines could be added the DAFS API.

Providing transparent memory registration to any legacy application requires to implement an automatic registration layer in ORFA. Actually, this is the same problem as the implementation of a MPI software layer. The MPI standard has been designed before the emergence of asynchronous network interfaces. Therefore, no memory management primitive has been included in the API [For94]. The application may thus pass any buffer to the MPI layer which cannot predict whether it is contiguous to previously used buffers, or will be used several times or not. This is the reason why most implementations such as MPICH-GM has to internally handle and optimize the registration of buffers passed by the application.

Therefore, the ORFA wrapper encloses remote file access by registration and deregistration routines. However, this method is too simple since several consecutive file accesses with the same buffer imply multiple registration and deregistration. Such a buffer usage is unfortunately very common. It was thus required to optimize this implementation to reduce the need of deregistration.

The first idea we used to fix this issue was to use a statically allocated registered zone as a temporary buffer. Copying data in this buffer is thus much faster than directly registering and deregistering it. However, this method is only suitable for small zones (up to 64 kB).

Actually, ORFA registration strategy was taken from MPICH-GM. It consists of delaying deregistration until it is really required (when no more pages can be registered). This acts as a registration cache since a previously used buffer will be ready much faster. Deregistration may be delayed until the end of the execution if possible. However, address space modifications have to be traced since they may invalidate entries in the translation table of the network interface. Therefore, the ORFA client also intercepts `mmap`-like functions (and several depending routines such as `free`) to force the real deregistration of pages whose address translation may be changed.

6 Conclusion

Our project ORFA was designed to use high-speed local area networks in order to connect user application buffers to a remote storage system in an efficient way. We focussed in this article on the implementation of our user-level ORFA client which provides a transparent access to remote data with support for complex UNIX functionalities. This issue has not really been discussed before and requires a good knowledge on shared libraries, symbols and dynamic linking.

Setting up a new custom API as DAFS offers a more specialized interface involving rewritten applications. This allows an efficient usage of the network architecture. However, we showed that this may be also achieved by legacy applications through a transparent shared library. The compatibility of the ORFA client with other libraries has been detailed while remote file access and manipulation is emulated to provide a full support for any complex functionality such as forking. The ORFA client may thus be used in both shell scripts and parallel applications without any modification.

Implementing our direct user-buffer access client also required to deal with memory registration. We showed that a registration cache coupled with address space modification tracing provides an efficient solution for a shared library client. Several performance evaluations has already

been published to show that such an implementation may saturate the physical link of MYRINET networks [GP03b] while the overhead of the ORFA wrapper remains negligible.

Nevertheless, the basic ORFA remote access library described in Section 3 may also be statically linked to any application. The similarity between ORFA remote access API and POSIX API would imply a very light rewriting work. This would provide an efficient access to remote files through the optimized network abstraction layer.

On the other hand, the transparency mechanisms detailed in Section 4 may also be used as an intercepting layer for several other projects such as GNOME VFS. This would make them available to any legacy application without rewriting and recompiling. The `dlfcn` method ensures compatibility with other libraries even when they modify standard routines. This may also be used in other context than remote file access or cluster environment.

7 Software Availability

The ORFA implementation may be downloaded from `http://perso.ens-lyon.fr/brice.goglin/work.html`. It is distributed under GPL license.

References

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[CLRT00] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[Clu02] Cluster File Systems, Inc. Lustre: A Scalable, High Performance File System, November 2002. `http://www.lustre.org`.

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[Fut01] William T. Futral. *Infiniband Architecture Development and Deployment*. Intel Press, August 2001.

[GP03a] Brice Goglin and Loïc Prylli. Design and implementation of ORFA. Technical Report TR2003-01, LIP, ENS Lyon, Lyon, France, September 2003.

[GP03b] Brice Goglin and Loïc Prylli. Performance analysis of remote file system access over high bandwidth local network. Research Report RR2003-22, LIP, ENS Lyon, Lyon, France, April 2003. Also available as Research Report RR-4795, INRIA Rhône-Alpes.

[MAF+02] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and Performance of the Direct Access File System. In *Proceedings of USENIX 2002 Annual Technical Conference*, pages 1–14, Monterey, CA, June 2002.

[MAFS03] K. Magoutis, S. Addetia, A. Fedorova, and M. I. Seltzer. Making the Most out of Direct-Access Network Attached Storage. In *Proceedings of USENIX Conference on File and Storage Technologies 2003*, San Francisco, CA, March 2003.

[Myr03] Myricom, Inc. GM: A message-passing system for Myrinet networks, 2003. `http://www.myri.com/scs/GM-2/doc/html/`.

[PcFH+02] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, 2002.

[PT97] L. Prylli and B. Tourancheau. Protocol Design for High Performance Networking: a Myrinet Experience. Technical Report 97-22, LIP-ENS Lyon, 69364 Lyon, France, 1997.

[SASB99] Evan Speight, Hazim Abdel-Shafi, and John K. Bennett. Realizing the Performance Potential of the Virtual Interface Architecture. In *International Conference on Supercomputing*, pages 184–192, 1999.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the Conference on File and Storage Technologies (FAST'02)*, pages 231–244, Monterey, CA, January 2002. USENIX, Berkeley, CA.