Design and Implementation of ORFA

Brice Goglin, Loïc Prylli

Design and Implementation of ORFA

Brice Goglin Loïc Prylli

September 2003

Abstract

ORFA is a user-level protocol that aims at providing an efficient remote file system access. It uses high bandwidth local networks and their specific API to directly connect user programs to data in the remote server. The ORFA model and its first implementation are described in this paper. The client is an automatically preloaded shared library that transparently overrides GLIBC I/O calls and redirect them to the server if concerned files are remote. The server may be a user program implementing a custom memory file system, or accessing native file systems.

Introduction

Achieving high performance for remote file access in a cluster environment require a high bandwidth data access to satisfy massive parallel applications running on MPI-IO, low latency for metadata access, and a usually strong coherency.

Existing projects generally focus on one of these important points by using stripping and parallezing or complex caching protocols. *Optimized Remote File System Access* is a new protocol that was designed for high performance general-purpose file access, making the most out of the cluster architecture, that is a high bandwidth and low latency local area network.

In this paper, we present ORFA design and describe its first implementation that is based on an automatically preloaded shared library client and a user-level server. We focus on ORFA protocol in Section 1, while the client architecture is discussed in Section 2 and the server in Section 3.

Contents

1 Remote File Access Protocol

ORFA is a user-level remote file access protocol. A user program accesses remote files through a user-level client that converts I/O calls into network requests, as described on Figure 1.

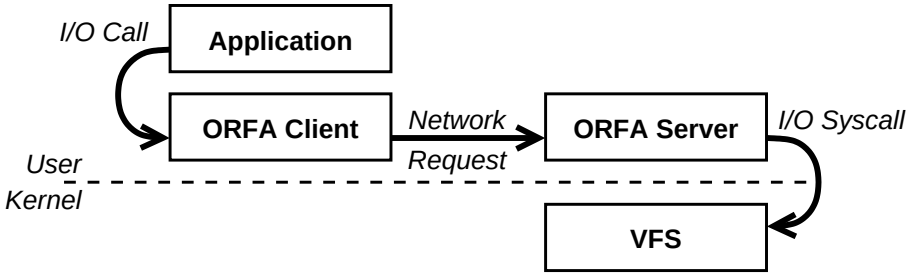


Figure 1: ORFA User-Level Remote File Access Protocol Model.

ORFA was designed for LINUX in a homogeneous cluster environment. Thus no XDR-like layer [RFC87] is provided to ensure compatibility among different architectures and operating systems. Nevertheless a 32/64 bits conversion is provided by the protocol, which is fully based on 64 bits data structures.

1.1 API Description

ORFA Request Type	*Target Object*
open, opendir, trunc, stat, lstat, statfs, mknod, mkdir rmdir, unlink, readlink, chmod, chown, lchown, access, utime	File Path
rename, link, symlink	2 File Paths
close, pread, pwrite, ftrunc, fstat, fstatfs, fchmod, fchown, flock, fsync, fdatasync, fcntl	FD ID
closedir, readdir, rewinddir	DIR* ID

ORFA Request Type	*Input Data*
open, opendir, trunc, utime stat, lstat, statfs, readlink mknod, mkdir, rmdir, unlink chmod, chown, lchown, access	File Path
rename, link, symlink	Merged Paths
pwrite	Data Buffer

ORFA Request Type	*Output Data*
fstat, stat, lstat	Inode Attributes
fstatfs, statfs	FS Attributes
readlink	Symlink Path
pread	Data Buffer

Table 1: ORFA Protocol Description. The first table presents targets of each kind of ORFA requests. Following tables shows whether a request requires an input and/or output buffer.

ORFA remains on the user-level file access API that is based on file paths, file descriptors and offsets. ORFA protocol was also built on main POSIX I/O calls. This model is not supposed to be used with caching clients. No `flush` or `revalidate` operation is available.

ORFA protocol is **not stateless** as NFS is [RFC95]. Opening a remote file or directory in the client links a virtual descriptor on the client to the real descriptor in the server through an identifier. Following calls will use this identifier as the target object (see Table 1).

Opening a file as a file descriptor with `open` (or as a `FILE*` through `fopen`) generates the use of a `FD` **identifier** on the ORFA protocol for following calls. However directory reading is handled in a special way, the `opendir` call creates a `DIR*` identifier.

This state-oriented model implies that the client may not handle a server failure properly. Actually on network or server error the client receives an error (usually `EIO`) and is supposed to discard its virtual descriptors.

ORFA clients can only handle seeking locally. All other I/O calls are supposed to be forwarded to the server, even descriptor manipulations such as changing open flags with `fcntl` or resetting a directory with `rewinddir`.

Any time an I/O call passes a user buffer as an argument, the corresponding ORFA request uses it as an input or output data for network requests. Table 1 summarizes requests need for such data.

1.2 Network Layer

ORFA was designed to use high bandwidth local network, especially Myrinet [BCF+95] (with GM [Myr00] or BIP software layer [PT97]). ORFA also provides a full TCP support through the socket layer.

1.3 Protocol Internals

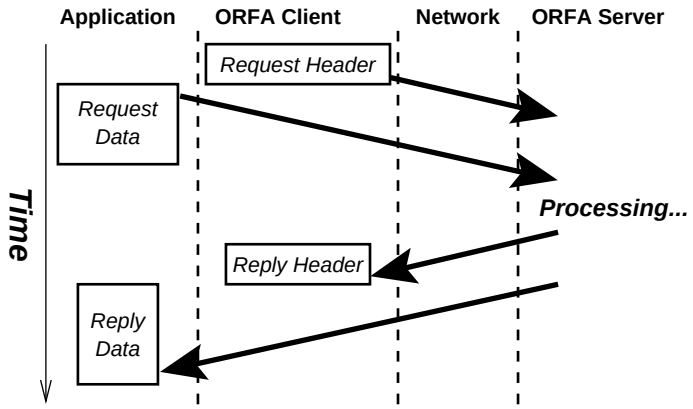


Figure 2: Message Passing Model used in ORFA

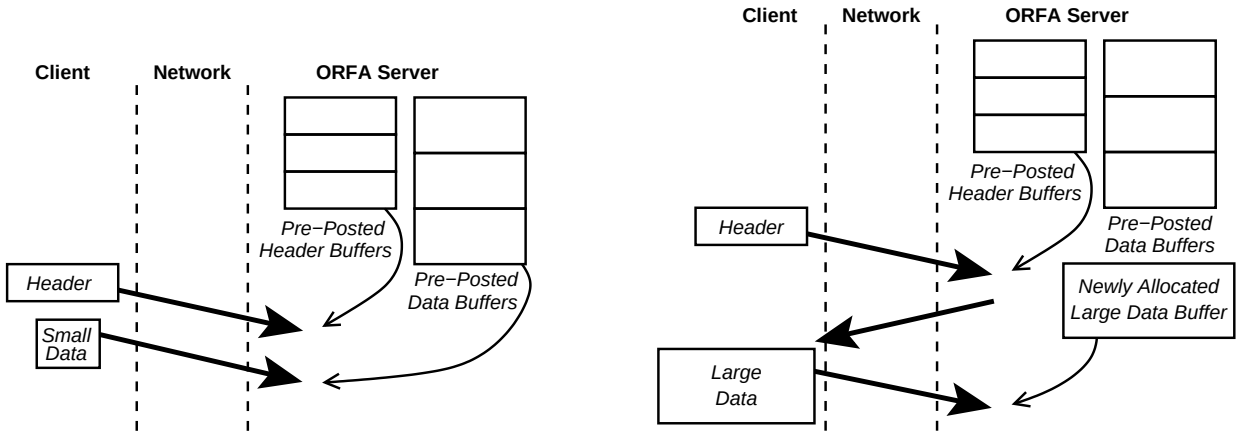
A ORFA request is composed of a header containing request parameters that may be followed by request data containing for example data to write or the file path to access. The reply is made of a reply header, eventually followed by a message containing for example read data or attributes.

Sending a memory buffer containing both request and associated data is faster than sending two distinct messages. However the input (or output) data buffer is given by the user program and is thus not contiguous to the ORFA header. Avoiding memory copy is important to maximize data transfer, especially for large buffers. That is the reason ORFA uses two distinct buffers instead

of trying to merge them using memory copies. A support for vectorized message in GM or BIP would solve this issue.

Thus the process is summarized in Figure 2. The client sends a first message, describing the request type, the target file, the context (user's uid/gid for example) and the size of the data if it exists. Then the data buffer is sent. The client receives from the server a reply composed of the result of the request (error code or bytes read for example) and the amount of data if it exists. Then the data buffer is received.

1.4 Pre-provided Buffers



(a) Small Buffer

(b) Large Buffer

Figure 3: Maximal Size Buffer Pre-Posting Model

The asynchronous network APIs we use (GM and BIP) need a receive buffer to be posted before a message arrives. As the server cannot predict what kind of request it will receive, it cannot predict how many receive buffers need to be pre-posted and what size they need to be.

Thus ORFA provides receive buffers in advance. The protocol sets a maximum size (generally 64 kB) and the server is supposed to pre-post enough header buffers or data buffers of this size for all messages it may receive (either headers or data).

If the client needs to send a smaller data, it can be sure a corresponding receive buffer has been posted on the server side and thus it can send safely (see Figure 3(a)).

If it needs to send a larger data, it waits for a confirmation message from the server. When the server receives the header, it notices that the amount of data to be received is higher than the size set by the protocol, and thus posts a larger buffer, and then sends a small confirmation message to the client (see Figure 3(b)).

This method maximizes the network utilization for limited data buffers. Only large buffers need to wait for a small message to be received, that is less than 10 μs, which is negligible compared to the buffer sending time.

It is important to see that letting clients try to send their header and input data buffer without any control will generate many missing receive buffers. This implies the lost of a large part of the throughput, that means poor performance.

2 Client Implementation

ORFA goals are similar to those of DAFS [Mag02], that is use the fastest way from the client to remote files, especially through the underlying high performance network. However the design is different, particularly on the client side. While DAFS defines its own specific API, which is fully asynchronous and also requires programs to be rewritten and recompiled, ORFA client tries to keep existing programs and add a transparent access to remote file systems.

2.1 Overview

The ORFA client is based on a shared library that intercepts I/O calls and redirects them to local files or remote servers. Assuming that physical networks available in clusters provide a very low latency (less than 10 μs), it has been designed as a lightweight client without intelligence. Each I/O requests is forwarded to the server and no caching is available. The client architecture is described on Figure 4 and is explained in following sections.

This drastically increases the number of small requests the ORFA client needs. For example, reading a directory usually requires a `readdir` request and a `stat` for each entry. This is huge compared to NFS, which gets lots of entries and their attributes through only one `READDIRPLUS` request.

To measure an example of metadata processing overhead, the ORFA client may use *read-ahead* when getting directory entries. Following `readdir` calls from the user application will be handled locally. This is the only case where the client shows intelligence. All other intercepted calls concerning remote files imply a remote call, that is a network request to the ORFA server.

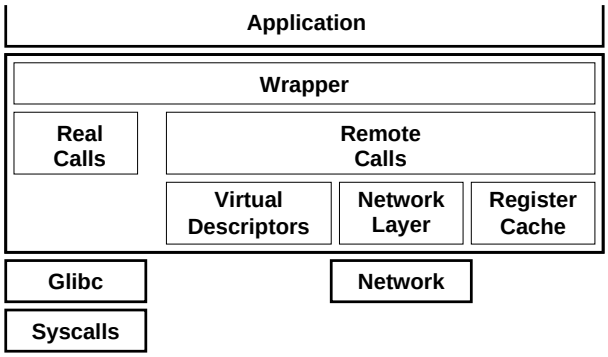


Figure 4: Architecture of the ORFA client.

2.2 Intercepting I/O Calls

Intercepting a function of the usual GLIBC library is done by defining a same name function in our library, and force its preloading. Setting `LD_PRELOAD` environment variable to the ORFA shared library client make it be loaded before all standard libraries. Dynamic linking then uses ORFA symbols instead of the GLIBC ones.

The ORFA shared client is organized around a main `wrapper.c` file, which contains redefinitions of I/O symbols (see the list in Appendix B). These symbols check whether the file is a local or a remote one and calls the corresponding sub-routine.

```
ssize_t read(int fd, void *buf, size_t count)
{
  if ( is_remote_fd(fd) )
    return remote_read(fd, buf, count);
  else
    return real_read(fd, buf, count);
}

void rewinddir(DIR *dir)
{
  if ( is_remote_dir(dir) )
    remote_rewinddir(dir);
  else
    real_rewinddir(dir);
}

int access(const char *pathname, int mode)
{
  char *new_path;
  int res;
  if ( is_remote_pathname(pathname, &new_path) )
    res = remote_access(new_path, mode);
  else
    res = real_access(new_path, mode);
  free(new_path);
  return res;
}
```

Local/remote check is done through `is_remote_*` functions. `is_remote_pathname` scans the full pathname, adding the current working directory, removing `.` and `..`, and checks for a `/REMOTE@server/` beginning (which is currently used to mean remote file on server `server`). This function copies the full-simplified path into a newly allocated buffer to reduce following parsing work.

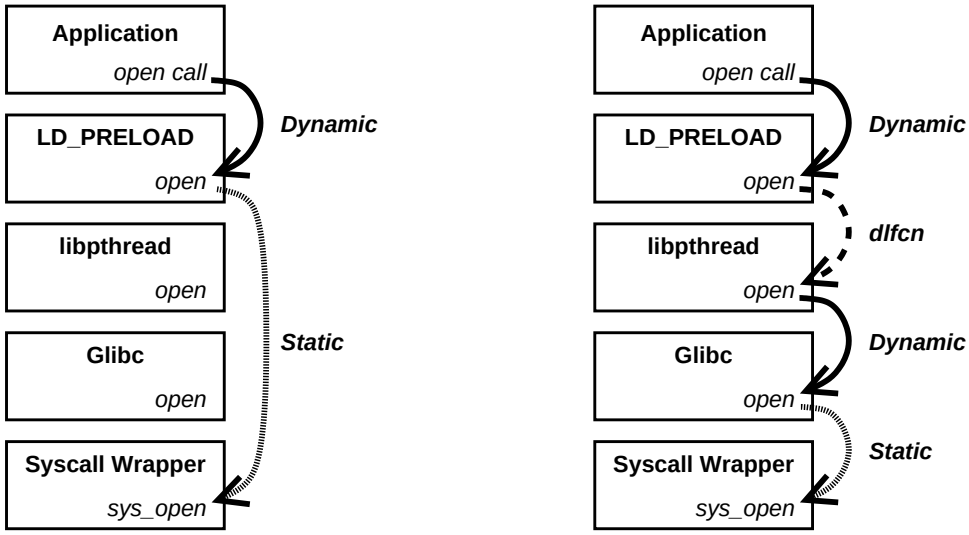
`is_remote_fd` and `is_remote_dir` functions check whether the given descriptor is a virtual descriptor associated with a remote file (see Section 2.5).

2.3 Finding Real Calls with `dlfcn`

When the target object is a local one, the wrapper function calls the real function of the GLIBC. Some previous work (such as `smbsh` or user-level PVFS implementation [CLRT00]) used a copy of the internal GLIBC code (generally just a syscall wrapper) to reproduce real calls behavior for local files. This prevents from supporting the evolution of the GLIBC code, and several special cases where another library (for example `libpthread`) is loaded before GLIBC to fix its behavior (see Figure 2(a)).

ORFA solves this issue by calling the real GLIBC function, which has been overridden. This is done through the `dlfcn` library, which provides a way to look for another occurrence of a symbol in the process address space.

```
real_fsync = dlsym(RTLD_NEXT, "fsync");
```



(a) Static linking with GLIBC code copy

(b) Dynamic linking with `dlfcn`

Table 2: Real Symbol Finding Models

This ensures a full transparency for local file access because the wrapper completely remains on usual GLIBC I/O calls (see Figure 2(b)).

2.4 Remote Calls

If the wrapper recognizes a remote file, it calls the dedicated routine, which prepares ORFA requests and then asks the network layer for data transfer.

```
int remote_ftruncate64(int fd, off64_t length)
{
  struct client_request req;
  fd_id_t fd_id;
  server_t server;
  /* get server and fd identifier */
  get_remote_fd_server(fd, &server, &fd_id);
  /* fill ORFA header */
  req.req.type = FTRUNC_REQ;
  req.req.data.ftrunc.fd_id = fd_id;
  req.req.data.ftrunc.length = length;
  /* call the network layer */
  network_call(server, &req);
[...]
}
```

As described in Section 1.1, the ORFA header is filled with its type and parameters. For file or directory descriptor oriented requests, the descriptor's identifier is copied into the header structure. For path-oriented requests, the application path buffer is given as an input data buffer to the

network layer. The same mechanism is used for other kinds of input and output data buffers.

```
  req.in_buf = in_buffer;
  req.out_buf = out_buffer;
```

2.5 Virtual File Descriptors

When accessing a file through its pathname, the localization of this remote file is described in this path (`/REMOTE@server/` references the root on the remote server `server`). However when using a file or directory descriptor, the ORFA client must be able to get its localization back. That is the reason each remote open file or directory is described by a local structure, a virtual descriptor.

```
struct file_descr {
  /* identifier */
  server_t server;
  fd_id_t fd_id;
[...]
};
```

Moreover this descriptor stores several details to reproduce the usual behavior during `lseek` or several special calls such as `dup` or `fork`.

```
  /* usual file descriptor fields */
  off64_t size;
  off64_t seek;
  /* use count */
  int use_count;
```

An efficient method to support all these special calls has been implemented by storing this structure into a memory-mapped file, whose file descriptor is given to the user application. Storing directory descriptors also uses some of these aspects.

This implementation provides a full and transparent support for manipulation of remote file descriptors through `fork`, `dup` or `exec`. Multithreading support is also available through protection of virtual descriptors against concurrent modifications.

2.6 Memory Registration

Any asynchronous network API requires buffers to be registered to the network interface and pinned in physical memory until the data transfer really occurs. This registration operation can be long on VM-oriented API such as GM. Thus memory registration needs to be used only when necessary.

ORFA client must use registered zones for headers and input/output data buffers. Instead of registering and deregistering each of these, for each I/O call, all small zones are copied into static zones that are registered during initialization. This additional memory copy has an overhead that is largely negligible against the registration it avoids.

Nevertheless a registration cache, which is generally more efficient, can replace this optimization. This cache consists of delaying deregistration until too much pages have been registered. This drastically reduces registration overhead when concerned pages are redundant. This argument is especially true for I/O applications because they generally use the same buffers for several consecutive data accesses. These buffers are registered only once.

2.7 Variant of ORFA Implementation on Client side using FUSE

Actually ORFA also provides the possibility to mount a remote file system in the kernel rather than using its `LD_PRELOAD` based shared library client (for cases that `LD_PRELOAD` cannot handle, for example, statically linked programs). Instead of porting the entire ORFA client as a kernel module, we based our kernel support on the FUSE project (File system in USEr-space) that allows connecting a file system implemented in a user program to the VFS layer. Mapping a remote file into a user program memory was the only function that was impossible to support with ORFA shared client, while mounting a remote file system with FUSE gives automatic kernel support for all Unix functionalities.

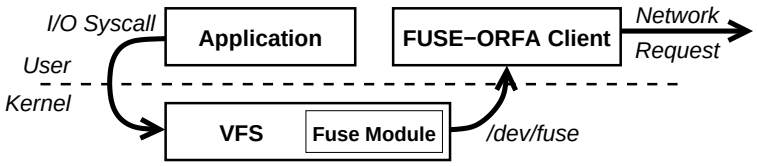


Figure 5: Kernel Mounting with ORFA and FUSE.

This implementation has the drawback of requiring data to go from the user program into the FUSE kernel module, and then back to user-space in the FUSE-ORFA client that communicates to the ORFA server. This implies an important latency and limits the bandwidth in the client because of the two memory copies (see Figure 5). These added costs have an impact only on the client side, so it is still useful for situations where the shared library approach is not suitable, but where we want to increase the scalability of the server side.

3 Server Implementation

Getting the most out of the underlying network layer requires ORFA to use the fastest way from clients to storage space. This implies removing layer redundancies and memory copies as often as possible. An interesting solution would have been to use a kernel server so that going from user-level to kernel would have been done during the network traversal.

We began our work on the server by using a user-level implementation which is simpler than a kernel one. We will mention in 3.1 several advanced subsystems on which this implementation relies. We also implemented two variants of this server. The first is based on a local file system on the server. The second is based on a custom memory file system (by removing the overhead of using syscalls and going through the VFS layer, this later variant allow to test the ORFA protocol independently of the storage subsystem of the server).

The ORFA server has to deal with all network requests coming from ORFA clients, process these requests, and send all replies. It has been designed to provide an experimental platform for testing several methods of handling all this work.

3.1 Events Handling

When using TCP network layer the ORFA server may have to deal with hundreds of connections, that means being able to get events from them in an efficient way. As the usual `poll/select` strategy is known to be not scalable to hundreds of clients, we also added support for the `epoll` and LINUX AIO API that provide much better performance.

Whatever network layer is used, the server then has to process many requests. The *sequential* architecture consists of receiving all pending requests, processing them all and then sending all answers back to clients. The ORFA server also provides a *queue-thread* architecture that uses one thread for receiving, one for processing and another one for sending. Moreover, it is possible a *connection-thread* architecture, that is to use a thread for each TCP connection.

Finally the server tries to make the most out of asynchronous API such as GM and BIP for network access, and also LINUX AIO for data access. This provides an efficient event-based architecture, even if full kernel support for disk AIO is still in development.

All these techniques provide a way to test the efficiency of several server architectures and find out which one is the most adapted to the workload the ORFA has to deal with.

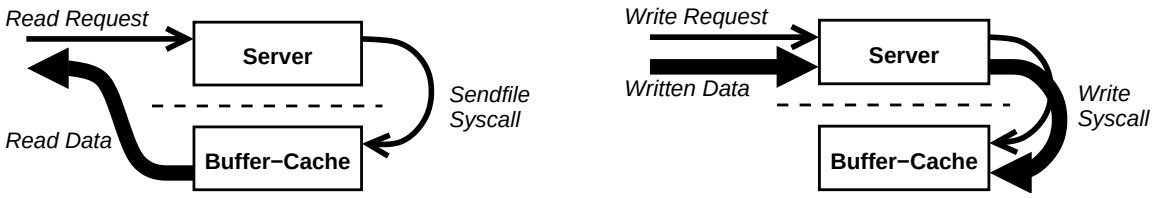
3.2 Storage Systems

3.2.1 Native File System

The main ORFA server converts incoming requests into I/O calls to the underlying native subsystems through syscalls. Data is stored as local files on the server host.

As described in the previous section, LINUX AIO is also used to make data access asynchronous, and thus increase the server reactivity. This also shares workload across different processors on the host (without using user-level threads)

This mechanism suffers from memory copies between user and kernel space. Thus `sendfile` is used to reduce per byte overhead on the server, when using TCP or BIP (see Figure 6(a)). As `sendfile` was developed to increase web-server efficiency, no `recvfile` implementation is currently available (we developed an experimental one in BIP but we do not use it). This is an important limitation concerning the memory copy avoidance (see Figure 6(b)).



(a) `read` request with `sendfile`.

(b) `write` request with a memory copy.

Figure 6: Path of data in the server during `read` (with `sendfile`) and `write` request.

3.2.2 User-Level Memory File System

Using native file systems require our user-level server to use a syscall and go across the VFS layer for each request. This implies a significant overhead compared to the total request processing time.

It is noticeable that data that is exported by the ORFA server do not necessarily require being accessible as local files on the server. We may imagine a physical storage space that would be dedicated to the ORFA server and not accessible in other way. This has the advantage of removing VFS overhead (used to protect data from concurrent modifications and provide support for several specific features, what is useless here).

Instead of implementing a new physical storage system in a dedicated partition, we simply developed a memory file system in the ORFA server. This explicitly removes the copy between user and kernel space, syscall overhead, and several other constraints.

This memory file system is implemented with a usual API while its storage organization is adapted to maximize ORFA protocol performance.

Conclusion

We described in this paper the protocol of ORFA, the client implementation as an automatically preloaded shared library and the user-level server. The lightweight client has been designed to use the fastest way to transfer data on the server side, while the absence of caching may imply lots of small network requests. The ORFA server wants to be as efficient as possible, using several techniques to reduce its CPU workload and increase its reactivity. Finally the implementation tries to make the most of the underlying high bandwidth local network.

This aims at providing a high performance remote file access, that means efficiency for parallel computing applications. However the full support the POSIX API on remote files and the client transparency make it available for any kind of interactive usage, or shell scripts. This contributes to the general-purpose file access ability of ORFA protocol.

The first version of the ORFA project produced user-level implementation of the client and server. We are now working on a second version where the client is implemented in the kernel directly under the VFS layer of the operating system. The server will still be user-level and the protocol is being changed to take advantage of the VFS layer properties.

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.

[Mag02] K. Magoutis. Design and Implementation of a Direct Access File system (DAFS) Kernel Server for FreeBSD. In *Proceedings of USENIX BSDCon 2002 Conference*, San Francisco, CA, February 2002.

[Myr00] Myricom, Inc. The GM Message Passing System, July 2000. `http://www.myri.com/scs/GM/doc/gm_toc.html`.

[PT97] Loic Prylli and Bernard Tourancheau. BIP Messages User Manual, 1997.

[RFC87] RFC 1014. XDR: eXternal Data Representation Standard, June 1987.

[RFC95] RFC 1813. NFS Version 3 Protocol Specification, June 1995.

A Usage

The ORFA implementation that is described in this paper is available as a tarball on `http://www.ens-lyon.fr/~bgoglin/work.html`. Source files are organized in `client` and `server` directories and their respective subdirectories, while common headers are centralized in `include`. When compiled, binaries are stored in the `bin-${ARCH}` directory.

The shared library client needs to be placed in the `LD_PRELOAD` environment variable. This enables its automatic preloading for all programs that will be run later. Files exported on server named **server** are then available under `/REMOTE@server/`. Setting this environment variable in a shell does not enable remote files for shell specific features (such as completion). Launching a new shell is necessary.

The main directory also contains a patch to be applied to FUSE 0.95 to convert its API to fit with ORFA protocol (see Section 2.7).

B Supported I/O Calls

Following functions, when available in the host GLIBC, are supported for remote files. Several functions support is provided through the interception of underlying functions that their GLIBC implementation is based on.

```
open, _open, __open, __libc_open
open64, _open64, __open64, __libc_open64
creat, creat64
close, _close, __close, __libc_close
read, _read, __read, __libc_read
pread, _pread, __pread, __libc_pread
pread64, _pread64, __pread64, __libc_pread64
write, _write, __write, __libc_write
pwrite, _pwrite, __pwrite, __libc_pwrite
pwrite64, _pwrite64, __pwrite64, __libc_pwrite64
lseek, _lseek, __lseek, __libc_lseek
lseek64, _lseek64, __lseek64, __libc_lseek64
llseek, _IO_file_seek
fopen, fdopen, fclose, fread, fwrite, fseek, feof
truncate, ftruncate, truncate64, ftruncate64
stat, __xstat, _stat, __stat
lstat, __lxstat, _lstat, __lstat
fstat, __fxstat, _fstat, __fstat
stat64, __xstat64, _stat64, __stat64
lstat64, __lxstat64, _lstat64, __lstat64
fstat64, __fxstat64, _fstat64, __fstat64
statfs, fstatfs, statfs64, fstatfs64
mknod, __xmknod, _mknod, __mknod
mkdir, rmdir, unlink, rename, link
symlink, readlink
access, euidaccess
chmod, fchmod, chown, lchown, fchown
```

```
fcntl, _fcntl, __fcntl, __libc_fcntl
flock, lockf
utime, utimes
fsync, _fsync, __fsync, __libc_fsync
fdatasync
sendfile, sendfile64
getdents, getdents64
opendir, closedir
readdir, readdir_r, readdir64, readdir64_r
seekdir, telldir, rewinddir
```

To ensure that the server acts on its files in the exact same way the client does, several functions have to be intercepted to in order to transmit the I/O context to the server.

```
umask
setuid, seteuid, setreuid, setresuid
setgid, setegid, setregid, setresgid
```

Allowing the use of a remote directory as working directory requires overriding usual behavior.

```
fchdir, chdir, getcwd
```

Keeping the POSIX behavior of file descriptors for remote files needs interception of several calls. `dup` and `fork` calls just have to increase usage counters. `exec` has to keep remote descriptor details during process reset and deal with *close-on-exec* flags.

```
fork, _fork, __fork, __libc_fork
dup, _dup, __dup, dup2, _dup2, __dup2
vfork, __clone
execve, fexecve, execv, execvp, execle, execl, execlp
```

Memory mapping explicitly requires to deal with the buffer cache. This is the reason ORFA shared client cannot support mapping of remote files (even if a read-only mapping support may be imagined when non-concurrent modification is ensured). Nevertheless `mmap`-like functions are intercepted to update register caching when the process address space is modified.

```
mmap, mmap64, mremap, munmap
msync, _msync, __msync, __libc_sync
```

The usual GLIBC `malloc` library uses `__mmap`-like functions to manipulate huge free memory zones. These functions are not exported by the GLIBC and thus cannot be intercepted. Therefore it was required to intercept memory allocating functions. This was done by including an entire memory allocation library in ORFA shared client.

```
sbrk
malloc, calloc, realloc, free
```