# File System Stacks in Clive

*Francisco J. Ballesteros*
*TR Lsub 15-1 9-Mar-15*

*ABSTRACT*

File systems in Clive are tools to access real files and system services. There are different file systems, to store data, to provide access to particular services, to access remote file systems, to cache them, to trace calls to them, and so on. For the system and all other system components, a file system is just anyone implementing the ZX file system interface. This interface is designed with network latency problems in mind, which is the common case when accessing the system through the network. The different file systems and file system tools can be combined and used together to provide features like file access, backups, history dumps, and performance measurements.

## Introduction

Clive is a new OS being built at Lsub to permit the construction of efficient cloud services. Besides its main target, its file services are already in use at our laboratory and provide interesting features.

In this report we briefly introduce the Clive file systems from the system point of view (and not from the programmer´s point of view.). The programmer's interfaces described in [1] are now obsolete and it is better to refer to [2] for a description of the programming interfaces involved, which are outside of scope from this report. The rest of this section describes just enough of the programmers interface to understand how the pieces described later fit together.

As any other service in Clive, interfaces follow a CSP-like mode where different system components exchange messages to interact. Messages are sent through channels, and components providing a service are mostly operating on channels to interact with the rest of the world. It is not a surprise that most Clive software is written in Go [3].

The interaction is actually similar to an RPC model of interaction. However, requests may consist of multiple messages (usually sent through a channel), and replies may also carry multiple messages.

As an example, taken from [2], this is the operation used to write a file in a file server:

```
Put(path string, d Dir, off int64, dc <-chan []byte, pred string) chan Dir
```

Here, calling `Put` is one of the things required to update a file. Data being written in the file is actually sent through the `dc` channel seen in the parameter list. The result status from `Put` is actually similar to a *promise* [4]. That is, the calling program may keep the resulting channel and is it as a promise to access the result of the operation. In particular, this result ( `chan Dir`) shows two interesting features in Clive interfaces:

1. The data resulting from the operation (a directory entry for the file after the `Put` has been

completed) is conveyed through a channel to the caller.

2. Any error status from the operation (should it fail) is reported as an error when user tries to receive from the channel and notices it cannot do so.

For example, this can be used to try to update a file:

```
datachan := make(chan []byte, 1)
datachan <- []byte("hi, there")
close(datachan)
dc := fs.Put("/foo", nil, 0, datachan)
resdir := <- dc
if resdir == nil {
     status := cerror(dc)
     ...
}
...
```

If the call to `Put` refers to a remote file server, it is packed and sent through a channel to the remote program. But, nevertheless, the interaction model is still the same. The input data is usually sent through a channel, and the results are retrieved from an output channel.

The important bits here are how the channels fit in the system interfaces, and not the particular programming language or model used to program a client or a server. What has been said suffices to understand the rest of the report.

**File System Stacks**

A file system is anyone implementing the file system interface described in [cite:2], or in the Clive's user manual that can be found in [5].

The main user of a file system is the name space, which maps a textual description of path-directory-entry entries to an interface that resolves names to directory entries. For example,

```
> NS='/ /
>>   /zx tcp!zxserver!zx
>> '
> lf /zx
```

defines first a name space, by defining the `NS` environment variable, and then runs the `lf` command to list files at `/zx`.

The command run-time library will initialize a name space (because it has been defined) and then try to find directory entries at `/zx` to list them.

As far as the name space is concerned, the `tcp!zxserver!zx` describes a directory entry mounted at `/zx`. Files at or under `zx` are discovered using a `Find` request, files are read using a `Get` call, and are written using a `Put` call, and so on.

Because the description of the directory entry at `/zx` is a network address, the name spaces uses the `Rfs` ZX file server that provides a local interface for a remote ZX file tree. The resulting system is as depicted:

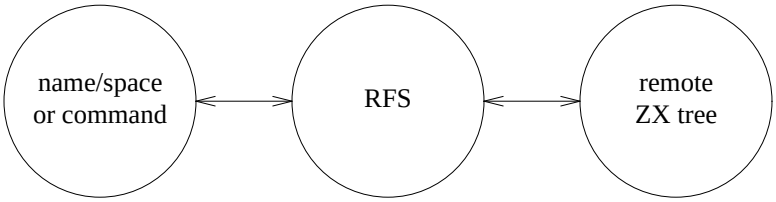


**Figure 1:** *RFS maps local ZX calls to remote ones.*

Thus, for example, when the user calls `Put`, a local (file tree) method call is issued to RFS, receiving the argument input channel for file data and returning the result output channel for both the final directory entry for the file and the error status. However, the implementation of the method now sends a message to a server for a remote ZX tree to issue the call, and then sends the input data from the input channel to the server, along with the call itself. As for the output, the reply message stream includes any data sent through the output channel and also the output status, which are delivered to the caller through the returned channel. This maps calls and channel I/O to network IPC nicely.

Now, in the remote machine we might have an actual ZX file server that relies on a disk to store files. That would be the `LFS` file system. As a result, the file system stack we are using is

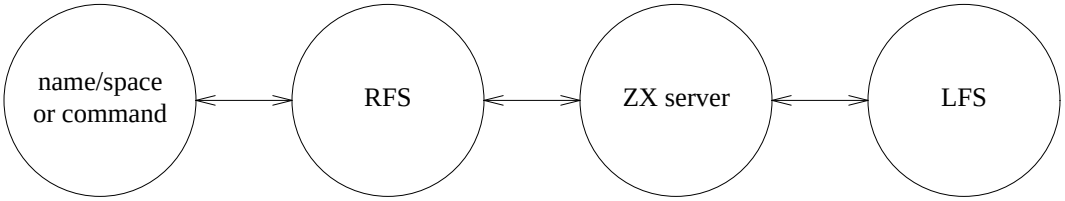


**Figure 2:** *File system stack when using RFS and LFS.*

The idea of stacking file systems is not new, and is as old as Plan 9, which is an ancestor of Clive using file interfaces to provide system services. However, Clive differs in two interesting ways:

1. Calls in the file system interface are designed to stream both input and output data.

2. Clive is more aggressive than Plan 9 was when stacking file systems.

As an example, this stack can be used to access a remote file tree, keeping a local on-disk cache for the remote files, and using a on-memory cache on the remote tree to cache the on-disk files on that machine. And to trace calls made to the client and the server tree.

Here

- TRFS accepts ZX calls and forwards them to another tree. For each call it posts a record of the description of the call to a channel, so the user could trace the calls.

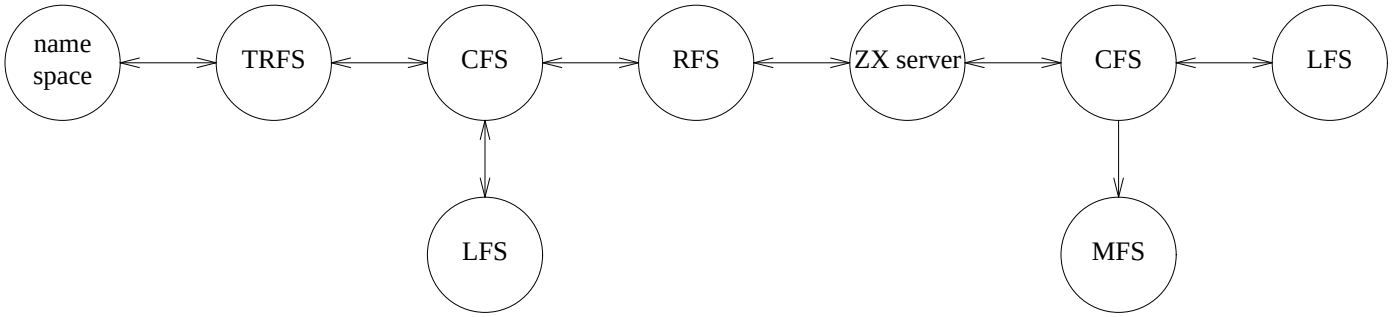


**Figure 3:** *Full fledged ZX file system stack using many file servers for caching and tracing.*

- CFS accepts ZX calls and relies on *two* ZX trees to serve them. It assumes the first one is a lot more cheaper than the second one, and uses the former as a cache of the latter. But, the two ZX trees used by CFS can be anyone.

- RFS maps local calls to a ZX server

- MFS serves files kept in RAM.

- LFS is a local file system and maps ZX call to files stored on disk.

And the interesting point is that we can combine all of them as we see it fits. For example, to trace calls to the cache in `CFS` and to the actual tree as used by `CFS` we might build this stack:

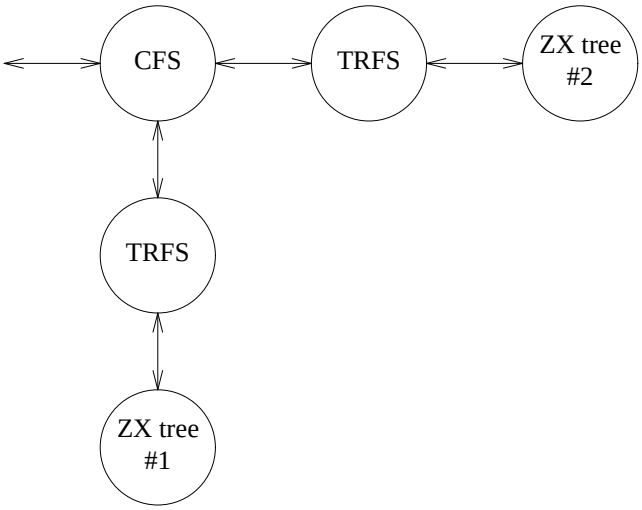


**Figure 4:** *Tracing calls to the cache and the cached trees in CFS using a file server stack.*

See for example this code:

```
zx1, err := lfs.New("a tree", "/foo", lfs.RW)
zx2, err := mfs.New("a cache")
tr1 := trfs.New(zx1)
tr2 := trfs.New(zx2)
cfs, err := cfs.New("a cached tree", tr1, tr2, cfs.RW)
err := <- cfs.Mkdir("/a/b", zx.Dir{"mode:": "0775"})
```

The first two lines build two ZX trees to use in the example (a local file tree and a on-memory tree). The next two ones stack tracing trees on each one. The fifth one creates a `CFS` cached file tree. The last one is an example to create a directory in the resulting ZX tree.

Should we want to trace also the calls to CFS, it is as easy as stacking yet another tracing file system before CFS:

```
trcfs := trfs.New(cfs)
...
err := <- trcfs.Mkdir("/a/b", zx.Dir{"mode:": "0775"})
```

We now described in a little bit more detail the few file trees that are implemented as of today. But we have to say that thanks to the simplicity of stacking file trees in clive, the number of file trees is quickly growing.

**Nested Control Requests**

When a ZX file tree operates by relying on another ZX tree deeper in the stack, it adapts its control interface to behave as a facade for it.

For example, the `/Ctl` file of a ZX file tree can be user to see the status of the debug flags, to see who is using the file tree (in the cases of trees served to the network) and to see usage statistics for the system.

This is the result of reading `/Ctl` on a stack made out of a `CFS` that uses a `MFS` as a cache, and a `RFS` as the remote tree. The `RFS` was connected to a remote server exporting a `CFS` that uses a `MFS` cache to serve a `LFS`. We have removed some of the lines when they do not help to further illustrate the case.

```
> cat /zx/Ctl
cfs:
fdebug off
vdebug off
noperm off
stat  8220 calls 252 errs 16440 msgs 0 bytes
     bgn: min    4.911µs  avg  625.628µs  max 1.333559776s
     end: min    5.045µs  avg  625.722µs  max 1.333559891s
get   3147 calls 0 errs 16501 msgs 38660497 bytes
     bgn: min   54.389µs  avg 2.657265ms  max 856.808309ms
     end: min   63.349µs  avg 5.141675ms  max 856.841591ms
...
```

```
cmfs:
debug off
noperm off
stat 78449 calls 2721 errs 156898 msgs 0 bytes
...

lsub:
user rfs:193.147.15.27:65348 nemo as nemo on 2015-03-26
user rfs:193.147.15.27:65349 nemo as nemo on 2015-03-26
ldebug off
rdebug off
stat 2656 calls 0 errs 5312 msgs 0 bytes
...


mfs:lsub:
debug off
noperm off
stat 47588 calls 2487 errs 95176 msgs 0 bytes
...

lsub:
debug off
rdonly off
noperm off
stat 2854 calls 0 errs 5708 msgs 0 bytes
...
```

Note the lines starting with `cfs:`, `cmfs:`, and `mfs:lsub:` . Only the first section comes from `Cfs`. Remaining lines come from `Cfs` reading the `/Ctl` file of the underlying ZX file tree and appending it to the data served to the user; and so on as we go down on the stack. Thus, the user may inspect the ZX stack as deep as it wants.

In the same way, we can set debug on for `Cfs` by executing

```
> echo debug on > /zx/Ctl
```

The ZX file server understands writes to the control file as textual commands to perform control requests.

But we can also set debug on the third tree in the stack:

```
> echo pass pass debug on > /zx/Ctl
```

When a control request starts with `pass`, this word is removed from the request and the server writes what remains of the control requests to the next tree in the stack. This makes the control interface for all the trees in the stack available to the final user.

File permissions and authentication may be used to adjust who can issue a control request to a ZX file tree, should that be a problem.

**Memory File Server**

`MFS` is a on-memory file server. It is similar to Plan 9's `ramfs`. Initially it starts as an empty tree (with the root directory), and accepts requests to create, read, write, and remove files and directories. The same streaming interfaces of any other ZX file tree are available.

This is an example of use:

```
// create a tree
fs, err := New("example mfs")
if err != nil {
    dbg.Fatal("lfs: %s", err)
}
dbg.Warn("fs %s ready", fs)
// Now use it...
```

**Local File Server**

`LFS` was the first ZX file system written. It provides ZX files using a local underlying file system (on disk). We use it to export alien file systems (eg., those from UNIX) to the rest of Clive.

This is an example:

```
var fs *Lfs // = New("tag", path, RO|RW)

dirc := fs.Stat("/ls")
dir := <-dirc
if dir == nil {
    dbg.Fatal("stat: %s", cerror(dirc))
}
dbg.Warn("stat was %s", dir)
```

Because ZX has directory entries (one per file) that have an arbitrary set of file attributes, `LFS` uses a separate file per directory to store those attributes when using an underlying UNIX file tree. But that is just how `LFS` is implemented.

**Memory-metadata, Disk Data File Server**

`MDFS` is a funny file server. It keeps the entire tree of directory entries on-memory, (like `MFS` does), but stores actual file data on a separate ZX tree. Thus, it always has to be used as part of a stack. For example

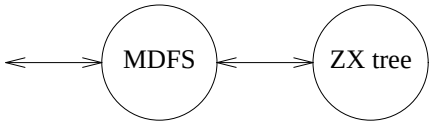


**Figure 5:** *MDFS keeps an on-memory tree for metadata for another tree.*

Usually, the next tree in the stack is only used by MDFS. But MDFS issues actual ZX calls to use it. That is, the file tree kept in that tree is exactly as seen by the MDFS client.

This is often used on away laptop machines as the cache tree for a CFS (yet another stack). The rationale is that we want the laptop to keep a copy of the cache on disk, should the connection to the server fail and should we want to use it as a simple LFS tree to survive the disconnection from the server.

On client machines that have highly available connections to the main server(s), using MFS as the cache for CFS suffices.

This is an example of use for MDFS:

```
// create a tree using the local directory /tmp/cache for the cache
dfs, err := lfs.New("cache", "/tmp/cache", lfs.RW)
if err != nil {
    dbg.Fatal("lfs: %s", err)
}
fs, err := New("example mdfs", dfs)
if err != nil {
    dbg.Fatal("mfds: %s", err)
}
dbg.Warn("fs %s ready", fs)
// Now use it...
```

**Caching File Server**

CFS, was shown before, provides a tree interface by relying on a tree used as a cache, and another tree which is the cached one.

As of today, no cache content is ever evicted. However, it would be easy to access directly the cache tree for CFS and simply remove what we do not want to keep in the cache by now. If the cache tree is a local LFS on a UNIX machine, we can just use the UNIX rm command to evict things from the cache.

If cache eviction proves to be necessary (it has not be the case for our usage of the system), we will put an evicting tree in the middle of the stack. And the eviction algorithm will become yet another file server that can be stacked with any other tree.

This is a (yet another) example for using CFS:

```
// create an in-memory tree
cachefs, err := mfs.New("example mfs")
if err != nil {
    dbg.Fatal("mfs: %s", err)
}
cachefs.NoPermCheck = true // cfs does this
// perhaps set debug for it
cachefs.Dbg = true

// and perhaps record the calls made to it
cachetrfs := trfs.New(cachefs)

// create an on-disk tree (OR or RW)
dfs, err := lfs.New("test fs", tdir, lfs.RW)
if err != nil {
    dbg.Fatal("lfs: %s", err)
}
dfs.SaveAttrs(true)

// create a cfs from them
fs, err := New("cfs", cachetrfs, dfs, RW)
if err != nil {
    dbg.Fatal("cfs: %s", err)
```

**Tracing File Server**

TRFS is a file server that forwards calls to another one and reports calls made. Reports are, of course, sent through a channel that is usually created by the caller:

```
type Flags struct {
    C    chan string
    Verb bool
    // contains filtered or unexported fields
}

type Fs struct {
    Tag string // set to prefix each msg with the tag.
    *Flags
    // contains filtered or unexported fields
}
```

Depending on the value of the `Verb` flag, `C` gets more or less call reports (eg., one per data message written to a file, or just one for the entire `Put` being performed on a file).

The format of the messages is fixed by TRFS, and other packages or uses might parse strings reported to perform more elaborate processing on traces. For example, CFS includes an elaborate tracing tool (built on TRFS) that tries to map calls made by the user to calls made to the cache and calls made to the cached tree.

**Using UNIX**

When using Clive from UNIX (which is a common case for us), there is a very important component in the ZX file system stack. At least it is important to be able to use UNIX commands on Clive's files.

We are referring to the `zxfs` command, which maps FUSE calls to ZX calls. For each UNIX (file system) call issued through FUSE, `zxfs` issues a ZX call to try to implement it using an underlying ZX file tree.

Needless to say that the underlying ZX file tree may be a full ZX file system stack.

**File Server Synchronization**

Although the ZX synchronizer is not part of the ZX file system stack, it relies on ZX trees just like everybody else. This is a command called `repl` used to keep replicas of ZX file trees.

The nice thing is that we can use `repl` to synchronize a local tree that was previously used as a cache of a remote tree, after using it directly. For example:

- We setup a `CFS` and use it to access a remote tree
- We shut down the network and start to use the `LFS` used as the `CFS` cache
- We bring up the network again
- We use `repl` to synchronize the changes locally made to the cache (that are not yet in the server)
- We set up again `CFS` to operate in an automated way.

Before locally using the cache as a file tree on its own, we would setup it as a replica of the remote tree with a command like

```
> repl -m cache examplecache /the/local/dir tcp!server!zx
```

Before using it again as a cache, we should synchronize again with the main tree:

```
> repl cache
```

**Name Spaces And User Operation**

Name spaces are not part of a ZX stack, strictly speaking. However, they are the main front-end for the user and for user commands to access ZX file trees. In any case, a ZX file tree may be accessed directly using its ZX interface, and it is not necessary to go through a name space. All trees are responsible for authenticating their users (if they want to) and performing permission checking (again, if they want to).

A name space does not provide a full ZX file tree interface. Its main purpose is to provide a `Find` call to find directory entries. To do so, it relies on remote trees and issues `Find` calls to them, similar to the `Find` call the name space implements.

The current implementation of a name space may be built out of a string describing the prefixes with mounted entries, and the list of entries mounted on each prefix.

When finding out which directory entries are present at (or under) a given path (and matching a given predicate), the name space is careful to adjust the predicates given to the `Find` calls sent to each of the underlying trees to the entries within a tree that are used as mount points in the name space are excluded from the search. The result is that although it is a prefix mount table, it works almost as a UNIX mount table would work, only that each mount point has a list of mounted directory entries (ie., is a union mount).

Another interesting feature is that, because the name space simply reports directory entries to its user, any directory entry may be mounted, and that does not need to refer to an actual file. Remember that in Clive a directory entry is simply a table of attribute names and attribute values, i.e., a `map[string]string` in Go.

The result is that the name space doubles as a registry besides being the name space.

As shown in the examples early in this report, changing the name space (or defining a new one) is as a easy as defining an environment variable. This does not make the system insecure, because the end file trees are always responsible for authenticating the users and performing any access checks required.

This is not as expensive as it might seem at first fight, because a command inherits the connections to remote trees from the parent process (a feature implemented in the Clive's `ql` shell), and not all processes have to build an entire set of connections to use them. Connections are dialed (and authenticated) on demand.

**Status**

Clive is too young and we are using it to continue developing it. Being a research system, we do not think twice when we find out that anything in the system may be done in a better way or may use a better interface.

For example, we are using the fourth implementation of `CFS`. The first one did not stack anything at all and was similar to the one found in Plan 9 (in spirit, at least).

Things are moving fast in clive and in the near future it is likely they will differ from what is reported there.

**References**

1. Clive's ZX file systems and name spaces Francisco J. Ballesteros Lsub TR/14/2. 2014. Available at [http://lsub.org/export/zx.pdf].

2. The Clive Operating System Francisco J. Ballesteros Lsub TR/14/4. Oct, 2014. Available at [http://lsub.org/export/clivesys.pdf].

3. The Go Programming Language The GO Authors See [http://golang.org].

4. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems B.Liskov, L Shrira. PLDI '88 Proceedings of the ACM SIGPLAN conference on Programming Language design and Implementation. 1988.

5. Clive User's Manual Lsub, 2015 http://lsub.org/sys/man