

# Clive's ZX file systems and name spaces

*Francisco J. Ballesteros*

## ABSTRACT

Clive is an operating system we are building that introduces a new file system interface and related utilities. The ZX file system leverages what we learned from Plan 9 but, at the same time supports streaming of requests and is amenable for CSP-like programming interfaces.

### File system interfaces

Before describing how the system manages remote file access, it is convenient to see the interfaces used to describe files and file trees. We show first the full interfaces and following sections describe the operations in more detail.

### File trees

This is a file tree as far a Clive is concerned

```
type Tree interface {
    // return a file from the fsys to operate on it.
    // The root name is "/". All other names are absolute
    // paths interpreted starting at the root.
    File(path string) (File, error)

    // similar to File(name).Stat()
    Stat(path string) (Dir, error)

    // Retrieve the contents of a file.
    // For directories, off and count refer to the number of
    // directory entries, counting from 0.
    // A count of -1 means "everything".
    // Each directory entry is returned as a series of
    // messages name:value, signaling the end of an entry with an
    // empty string.
    Get(path string, off, count int64, dc chan<- []byte) error

    // Like Get, but return messages written using PutMsgs
    GetMsgs(path string, dc chan<- []byte) error
}
```

As it can be seen, there are no methods to update the file tree. Trees capable of accepting updates are known as *Recver* trees:

```
// A Tree that can be subject of Recv()
type Recver interface {
    Tree

    // Update a file with d and data from dc.
    Put(path string, d Dir, dc <-chan []byte) error

    // Create a directory.
    Mkdir(path string, d Dir) error
}
```

Which are further extended to full read-write file trees:

```
// A full read/write tree
type RWTree interface {
    Recver

    // Update metadata
    Wstat(path string, d Dir) error

    // Delete the file or directory.
    RemoveAll(path string) error

    // Update/create a file with msgs from dc.
    PutMsgs(path string, d Dir, dc <-chan []byte) error

    // Move a file or directory
    Move(from, to string) error

    // Delete the file or empty directory.
    Remove(path string) error
}
```

## Files

A file is described by the following interface(s):

```
type File interface {
    // Return the fsys for this file
    Tree() Tree

    // Return the path for the file in its tree.
    Path() string

    Walker
}
```

Where a *Walker* is either a “Texas Ranger” or something similar to a file capable of accepting calls to let you walk the tree rooted at it:

```
type Walker interface {
    // Stat a file.
    Stat() (Dir, error)

    // Walk to a child
    Walk(elem string) (File, error)

    // Retrieve the contents of a file.
    // For directories, off and count refer to the number of
    // directory entries, counting from 0.
    // A count of -1 means "everything".
    // Each directory entry is returned as a series of
    // messages name:value, signaling the end of an entry with an
    // empty string.
    Get(off, count int64, dc chan<- []byte) error
}
```

The file interface is further extended to include files with full read-write access:

```
type RWFile interface {
    File

    // Delete the file or empty directory.
    Remove() error

    // Delete the file or directory, empty or not.
    RemoveAll() error

    // Update metadata
    Wstat(d Dir) error

    // Update/create a file with data from dc.
    Put(d Dir, dc <-chan []byte) error

    // Create a directory.
    Mkdir(elem string, d Dir) error
}
```

### White outs

Some trees (like replicated trees relying on a database of updates) might provide whited out entries at directories, and are expected to implement these other interfaces:

```
type WhiteOutTree interface {
    Tree
    WOFile(name string) (WhiteOutFile, error)
    WhiteOut(path string, mtime int64) error
}

type WhiteOutFile interface {
    File
    WOGet(off, count int64, dc chan<- []byte) error
    WOWalk(elem string) (WhiteOutFile, error)
}
```

## Directory entries

There are several important details in the interfaces shown so far. First, directory entries are a set of attribute names and values, as defined by

```
type Dir map[string]string
```

By convention, some attributes are expected to be defined, for example:

```
/*
    Conventional attribute names
*/
const (
    Size = "size" // number of entries in directories, size for files
    Name = "name" // file name
    Mtime = "mtime" // mod time since epoch (ns)
    Mode = "mode" // permissions (octal, 0777 bits must be unix bits).
    Type = "type" // dir/file/... (first char of ls -l listing)
    Rm = "rm" // whiteout (if set, the file is removed)

    Tfile = "-" // file is a plain file
    Tdir = "d" // file is a directory
)
```

Attributes that are understood as integer values (or time values) are encoded with using the “%d” verb of *Printf* and decoded as expected. Times are always expressed as the UNIX time in nano-seconds.

The extensible nature of directory entries has proven to be very useful. For example, when sending directory entries to other processes as a stream it may be useful to add a *path* attribute recording the full path for each directory entry. The receiver can rely on that to operate on the file described by the entry. The same can be done to add hashes for file content, etc.

Nevertheless, file systems are not expected to be able to store all the attributes defined by a directory entry. Thus, applications rely only on well-known (internet-wide) attributes to be kept properly in the store. Other attributes might vanish when the file is stored depending on the actual file system used.

The *Dir* data type defined *Send* and *Recv* methods used to send and receive directory entries through a *[]byte* channel. These are used along with the facilities described elsewhere [1] to send and receive them through channels connected to external pipes or connections.

The interfaces shown before include calls like these ones to retrieve the directory entry for a file and to update it.

```
// Stat a file.
Stat() (Dir, error)

// Update metadata
Wstat(d Dir) error
```

Usually, most operations are available both in the *Tree* interface (with the first argument indicating which file is affected) and in the *File* interface (with the file being implied by the object receiving the call). When updating the metadata only those entries present in the entry are updated.

## Read and Write

There is no *read* or *write*. Instead, there are *get* and *put* operations. This is the interface for *get*:

```
Get(path string, off, count int64, dc chan<- []byte) error
```

As it can be seen, the data retrieved will flow through the channel given as an argument. Of course, such channel might be closed with an error indication if the system process supplying the data encounters an error, or, it might be closed by the process retrieving the data if no more data is wanted. Considering the tools described at [1], it is trivial to stream the data to a pipe or a network connection.

The *get* operation works also on directories. In a directory, the data retrieved is a set of directory entries, plus an empty message (0-sized). Each directory entry is encoded as implemented by the *Dir.Send* operation (as a UTF-8 string indicating names and values for attributes defined; with quoted values).

We now show *put*:

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

As seen, a directory entry is supplied. That is useful when the caller wants to update the directory entry besides updating the file data. This is most useful in two cases:

- 1 When creating the file, to specify the metadata besides the data.
- 2 When updating the file, to specify a modification time for the file besides its data. This one is used by a file synchronization tool.

Like happen to *get*, *put* relies on a stream of data. All data send through the given channel, *dc*, will be put into the file. This channel might be closed by the system process receiving the data upon errors, or by the sender process. Once again, data might be streamed across the network to be written to a file.

As an aside, we are updating the interface for *put* to be

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

so that it could support insertions and removals at arbitrary places in a file. But, as of now, the one implemented is the one shown above.

As of now, we have an implementation for a local file system and another for a data base of file metadata used for file replication.

## File tools

There are several tools built upon file interfaces that can work on any resource implementing them. This function reads everything from the file:

```
func GetAll(f File) ([]byte, error)
```

For directories, this function reads everything and “unpacks” directory entries:

```
func GetDir(f Walker) ([]Dir, error)
```

Besides these and other tiny tools, there are more interesting ones:

```
func Send(f File, c chan<- []byte) (rerr error)
```

sends an entire file tree (that rooted at the given file) through the given channel. The function simply retrieves the directory entries (and data for files) and delivers all that through the channel. The end of file data is signaled as an empty message (as it happen to directory data streams). Its counterpart,

```
func Recv(fs Recver, c <-chan []byte) error
```

can be used to receive an entire file tree from the given channel.

For example, although the local file system knows nothing about the network or pipes, and knows nothing about how to send or receive a tree through channels, it is trivial to send a copy of a local file tree across the network:

```
// Provide a local file system for the underlying file tree at tdir.
fs, _ := NewLFS(tdir)
// Dial a network address and return an output channel to send data.
c, _ := ds.Dial("tcp!machine!svc", nil)
// c.Out is the output chan[]byte
root, _ := fs.File("/")
// Stream the entire tree.
Send(root, c.Out)
```

And it is as easy to receive the tree and create it at another place:

```
fs2, _ := NewLFS(tdir2)
// c is a connection with an input channel. See [1].
Recv(fs2, c)
```

Needless to say that data is streamed and latency does not suffer due to RPCs.

### Remote file access

But there are RPCs if you want them. Or, more precisely, there are RPCs relying on promises for their output results.

The data used as part of a *put* or *get* request is already dealt with by the channels used to stream it. The remaining problem is how to avoid the RPCs for calls made, while still providing RPCs for those that need them.

This is the interface for a tree to be used across the network:

```
type Tree interface {
    // stat a file
    Stat(rid Rid) chan zx.Dir

    // Retrieve the contents of a file.
    // For directories, off and count refer to the number of
    // directory entries, counting from 0.
    // A count of -1 means "everything".
    // Each directory entry is encoded as a series of "name=value"
    // pairs and the end of the entry is signaled with an empty string.
    // The chan is not closed at the end.
    Get(rid Rid, off, count int64, dc chan<- []byte) chan error

    // Retrieve messages from a file.
    GetMsgs(rid Rid, dc chan<- []byte) chan error
}
```

It is further extended for trees providing full read-write access

```
type RWTree interface {
    Tree

    // Update a file with d and data from dc.
    Put(rid Rid, d zx.Dir, dc <-chan []byte) chan error

    // Update/create a file with msgs from dc.
    PutMsgs(rid Rid, d zx.Dir, dc <-chan []byte) chan error

    // Create a directory.
    Mkdir(rid Rid, d zx.Dir) chan error

    // Move a file or directory
    Move(from, to Rid) chan error

    // similar to File().XXX()
    Remove(rid Rid) chan error
    RemoveAll(rid Rid) chan error
    Wstat(rid Rid, d zx.Dir) chan error
}
```

The main difference with respect to the interfaces shown early in this paper is that the values returned by the methods are always channels. That is, to perform an RPC, the caller must do something like

```
if err := <-rfs.Mkdir(mks[i].path, mks[i].d); err != nil {
    Fatal("mkdir failed with sts %v", err)
}
```

Note the receive before the call. Because of this receive, it is feasible to issue several calls and receive their results later on, which permits streaming requests and defer the reception of their replies until later, to save latency.

A couple of adaptors bridge the gap between these interfaces and the ones intended for local access. We tried first with a single interface to “rule them all”, but all the versions we tried were inconvenient in many cases. The set of interfaces presented here has been more convenient in practice, at least for us.

## Name spaces

Clive separates name spaces from names used in files. A name space is one out of two interfaces depending on whether you are just using the name space (finding names through it) or updating it.

A *finder* lets you navigate a name space:

```
type Finder interface {
    Find(path, pred string, depth0 int) <-chan zx.Dir
}
```

Here, a path is a file path to indicate the root of the name space (sub)tree where you want to start navigating. A *pred* is a predicate encoded in a string to select which directory entries are of interest for you. An empty string indicates all entries rooted at the given path.

But there are more interesting predicates. For example,

```
~ name "*.c" & depth < 3
```

would retrieve all entries for files (or whatever they might be!) matching the given expression for the name, not looking deeper than 3 levels down from the specified root.

As another example, reading a single directory is a matter of indicating a shallow depth. Walking directly to a single file can be achieved by indicating zero as the depth.

Note that the extensible nature of directories, plus the genericity of the operators defined for predicates, make *find* a very powerful tool. It can be used as conventional lookups in the name space, but it can be used to select just those directory entries of interest. The caller might then start retrieving the stream of directory entries and for each one issue a (perhaps streamed) request to the process holding the file.

That is, things like removing for all object files within a tree can now be done in two round trips. And in fact, due to the pipe-line effect, it might appear to be just one.

The name space is actually a mixture of a prefix mount table and the Plan 9 mount table. Or, to be more precise, it is a reimplementaion of the ancient Plan B mount table reworked for Clive.

Updating entries is done through the *binder* interface:

```
const (
    Repl Flag = iota
    Before
    After
)
type Binder interface {
    Mount(name string, d zx.Dir, flag Flag) <-chan error
    Unmount(name string, d zx.Dir, suffixes bool) <-chan error
    Bind(from, to string, flag Flag) <-chan error
}
```

But this is not as interesting as the traversal described above, and is not further discussed here.

**Acknowledgements**

We are very grateful to Roger Peppe and Charles Forsyth for their insights and help.

**References**

1. Pipes, connections, channels and multiplexors. Francisco J. Ballesteros. Lsub TR/14/1.