The Clive Operating System

Francisco J. Ballesteros
TR Draft Lsub 14-4 - 4-Oct-14

ABSTRACT

Clive is an operating system aimed at providing simple and high performance construction of network and cloud services. Currently it is used hosted on other systems to support further development of the system. In the future it will also run on native blades. Clive introduces new file system technology and novel interfaces for the construction of networked system. This paper describes most of the system, written in Go using a CSP style.

Introduction

Clive is a new system being built at Lsub leveraging two previous ideas:

- CSP and descendant languages (including Go). A programming style for concurrent programs based on independent processes communicating through channels. This style fits well with distributed programs and can help to address network latency problems.
- Many software stacks in blades used for cloud computing are being used just to run software in a single language. This is most common for Java, but can be the case for others. The Nix OS shows that applications can run without interference from the OS in machine cores. Thus, we could make efficient cloud services written in Go by running them on the bare machine.

In short, we are modifying the Go compiler and runtime to provide better interfaces for network programming and to be able to cross-compile applications written in Go so they are their own kernel and can run without any OS.

The resulting system is designed both to

- run on native blades with high performance
- run on hosted (alien) operating systems to leverage them as terminals and servers

In this paper we introduce the system and describe the interfaces designed, the related services as provided by Clive and how to write network applications using them.

Clive For Users

Before proceeding any further, here are some usage examples. The first example shows that listing all sources files under `/zx/sys/src` can be done with this command

```
; lz /zx/sys/src/,
/zx/sys/src
/zx/sys/src/clive
/zx/sys/src/clo
```

Here, describing a file (or a set of files) is done by using a combination of a name and a predicate. In this case, all files starting at `/zx/sys/src` and matching an empty predicate (the empty string) are listed.

Or we can list only directories using

```
; lz /zx/sys/src/,type=d
```

Or we can remove all regular files in a hierarchy by using

```
; rm /zx/sys/src/,type=-
```

Commands operating on files find the involved directory entries and then operate on them. A directory entry is a set of name/value pairs, and may have any number of attributes with names and values chosen at will. Of course, there are some conventions about which attributes are expected to be there (like size, mode, etc).

Predicates used to find files are general expressions that may use directory attributes as values, which makes it easy for a command to issue an expression to find the entries of interest in a single or a few RPCs.

Directory entries are self-describing entities (eg., they report also the address of the server and the name of the resource in the server). This makes it easy for a program to issue requests for a directory entry it found.

In short, file trees in clive are split into two important entities:

- Finders used to find directory entries
- File trees that accept operations for directory entries

Each process groups one or more finders into a name space, built from a textual representation (it might inherit the name space from the parent).

For example, we can use

```
; NS='/ /
;;   /zx tcp!zxserver!zx
;;   /dump tcp!zxserver!zx!dump
;; '
; lz /zx/usr/nemo,type=d
```

to define a new name space and then issue commands that work in it. In this example, we defined as `/` the root of the host OS file tree, and then mounted at `/zx` our main tree and at `/dump` its dump file system.

To say it in a different way, the name space is a finder that may groups other finders (among other things). The name space is more powerful, and can mount at a given name a set of directory entries (be they for files or not), but the example suffices for now.

Clive For Programmers

The next example shows that the `lz` example above can do its work by relying on the `Find` call of a finder, as defined by [clive/nspace]. This interface can be used to access a stream of directory entries matching the `Find` request.

```
dirc := ns.Find("/zx/sys/src", "type=d", "/", "/", 0)
for dir := range dirc {
     if long {
          fmt.Printf("%s\n", dir.Long())
     } else {
          fmt.Printf("%s\n", dir["path"])
     }
}
```

The `Find` request issued a call to the name server and resulted in a channel from where we can receive directory entries as they are sent to us. Note the implication for issues related to the network latency.

The next example shows how the `rm` command can remove all these files by working almost in exactly the same way, but issuing calls to remove the files instead of just printing the directory entries:

```
errors := []chan error{}
for dir := range dirc {
     // get a handle for the dir's tree
     wt, err := zx.RWDirTree(dir)
     if err != nil {
          dbg.Warn("%s: tree: %s", dir["path"], err)
          continue
     }
     errc := wt.Remove(dir["spath"])
     errors = append(errors, ec)
}
for _, errc := range errors {
     if err := <-errc; err != nil {
          dbg.Warn("%s: %s", dir["path"], err)
     }
}
```

This example shows another important thing: we can send streams of requests to all involved servers (some server might be responsible for multiple directory entries received by the find request), and then receive their output (in this case, an error indication for each one).

Channels can be used very much like promises or futures. Furthermore, if the error channels seen are buffered, and they might be just ignored if we are not interested in the replies.

Channels In Clive

Clive system services are organized by connecting them through a pipe-like abstraction. Like it has been done in UNIX for decades. The aim is to let applications leverage the CSP programming style while, at the same time, make them work across the network.

The problem with standard Go (or CSP-like) channels is that:

1. They do not behave well upon errors, regarding termination of pipelines.

2. They do not convey error messages when errors happen.

Therefore, we modified the channel abstraction as provided by Go to make it a better replacement for traditional pipes. When using channels in Clive's Go, each end of the pipe may close it and the channel implementation takes care of propagating the error indication to the other end. Furthermore, an error string can be supplied when closing a channel and the other end may inquire about the cause of the error. This becomes utterly important when channels cross the network because errors do happen.

For example, consider the pipeline

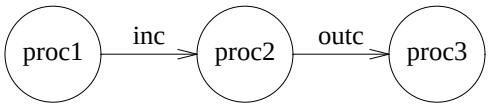

Figure 1: *Example pipeline of processes in clive*

In Clive, `proc2` can execute this code to receive data from an input channel, modify it, and send the result to an output channel:

```
var inc, outc chan[]byte
     ...
for data := range inc {
     ndata := modify(data)
     if ok := outc <-ndata; !ok {
          close(inc, cerror(outc))
          break
     }
}
close(outc, cerror(inc))
```

Should the first process, `proc1` , terminate normally (or abnormally), it calls `close` on the `inc` channel shown in the code excerpt. At this point, the code shown for `proc2` executes `close(outc, cerror(inc))` , which does two things:

1. retrieves the cause for the close of the input channel, by calling `cerror(inc)`
2. closes the output channel providing exactly that error indication, by calling `close` with a second argument that provides the error.

Therefore, the error at a point of the pipe can be nicely propagated forward. In its interesting to to reconsider the implications of this for examples like that shown for removing files, and for similar system tools.

The most interesting case is when the third process, `proc3` , decides to cease consuming data. For example, because of an error or because it did find what it wanted. In this case, it calls `close` on the `outc` channel shown in the code.

The middle process is not able to send more data from that point in time. Instead of panicing, as the standard Go implementation would do, the send operation now returns `false` , thus `ok` becomes `false` when `proc2` tries to send more data. The loop can be broken cleanly, closing also the input channel to signal to the first process that there is no point in producing further data.

Furthermore, all involved processes can retrieve the actual error indicating the source of the problems (which is not just *"channel was closed"* and can be of more help).

As an aside, the last call to `close` becomes now a no-operation, because the output channel was already closed, and we don't need to add unnecessary code to prevent the call because in Clive this does not panic, unlike in standard Go.

The important point is that termination of the data stream is easy to handle for the program without resorting to exceptions (or panics), and we know which one is the error, so we can take whatever measures are convenient in each case.

Networked Channels

For communication between processes that might cross the network or to reach external devices, we rely on channels of slices of bytes (in what follows we use the term *device* to refer to an external file descriptor, pipe, or network connection in general).

The [nchan(2)] package provides tools that help processes to use channels and mostly forget about external devices (eg., network connections). For example, consider

```
func ReadMsgsFrom(r io.Reader, c chan<- []byte) (nmsgs int64, nbytes int64, err error)
    Send everything read from r to c, preserving message delimiters. The
    buffers sent through c can be given to bufs.Recycle to reuse them when
    no longer needed. The chan capacity determines how many buffers are
    allocated.

    If an error was sent to the device we are reading from (eg., by closing
    the chan given to WriteMsgsTo), it is retrieved and the c is closed with
    the same error.
```

and

```
func WriteMsgsTo(w io.Writer, c <-chan []byte) (int64, int64, error)
    Write everything received from c to w, preserving message delimiters.
    Does not stop on an empty write. if c is closed with an error
    indication, the error is sent through the writer.

    Returns the number of messages, bytes, and error.

    When Buffering is false, messages sent are written directly.

    When Buffering is true, if w implements Flusher, w.Flush() is called
    after each message, unless w implements DontFlusher (which means that
    the caller is handling buffering itself).
```

Using these, code like the one shown in the example above for `proc2` can work correctly even if the input (or output) channel comes from a pipe or network connection and not from another process within the same program. These functions are built by relying on, more simple

Messages are preserved through the underlying device, and each one is sent indicating the actual message length. There are more tools in `nchan` that do not delimit messages and handle raw streams of bytes, but most of Clive relies on channels delivering messages.

Upon errors, a peculiar message length is sent that indicates that the next message indicates an error string and not a data message. Of course, if the error message cannot be sent because the network is broken, the affected end of the connection supplies a more generic *IO error* message.

These tools are seldom used in favor of higher level tools, like

```
func NewConn(rw io.ReadWriter, nbuf int, win, wout chan bool) Conn
    Like NewSplitConn(rw, rw, nbuf, wout).
```

where

```
type Conn struct {
    Tag string // debug
    In  <-chan []byte
    Out chan<- []byte
}
```

A `Conn` joins two channels to make a full-duplex connection. A process talking to an external device relies on this structure to talk to it. The `Tag` field can be used to record the address or the name of the other end of the connection, for debugging or any other purpose.

Direct piped connections can also be make:

```
func NewConnPipe(nbuf int) (Conn, Conn)
```

This is very useful for testing, because the connection can be created with no buffering and it is easier to spot dead-locks that involve both ends of the connection. Once the program is ready, we can replace the connection based pipe with an actual system provided pipe.

Multiplexed And Streamed RPCs

The `nchan` package provides multiplexors built upon the tools described above.

```
type Mux struct {
    In      chan Conn
    Tag    string
    Debug bool
    // contains filtered or unexported fields
}
   Muxes a Conn so that there can be multiple in/out channels within the
    same connection. One side of the connection is said to be the caller and
    the other the callee, this is indicated when the Mux is created.
```

In the multiplexed connection requests may carry a series of messages (and not just one message per request) and may or not have replies. Replies may also include a full series of messages. Both ends of a multiplexed connection may issue requests. Thus, this is not a client-server interaction model, although it may be used as such. For example, to issue a outgoing request (with no expected reply) through the multiplexor, we can execute this:

```
oc := mux.Out()
oc <- []byte("no reply")
oc <- []byte("expected")
close(oc)
```

To issue a request with an expected reply (i.e., and RPC), we can instead execute:

```
rc, rr := mux.Rpc()
rc <- []byte("reply expected for this")
rc <- []byte("request")
close(rc)
for m := range rr {
     Printf("got %v as part of the reply\en", m)
}
Printf("and the final error status is %v\en", cerror(rr))
```

Of course, this can be done multiple times to issue several concurrent outgoing requests. This interface permits both the RPC and the streaming styles of interaction.

The interface for the receiving part of the multiplexor is a single `In` channel that conveys one `Conn` per incoming request. The request has only the `In` channel set if no reply is expected, and has both the `In` and `Out` channels set if a reply is expected. The code might look like this:

```
for call := range mux.In {
     // call is a Conn
     for m := range call.In {
          Printf("got %v as part of the request\en", m)
     }
     if call.Out != nil {
          call.Out <- []byte("a reply")
          call.Out <- []byte("was expected, but...")
          close(call.Out, "Oops!, failed")
     }
}
```

When `mux.Out` is used, the multiplexor works using a `Conn` with just the `Out` channel. For example, for two of such requests this figure depicts the result.

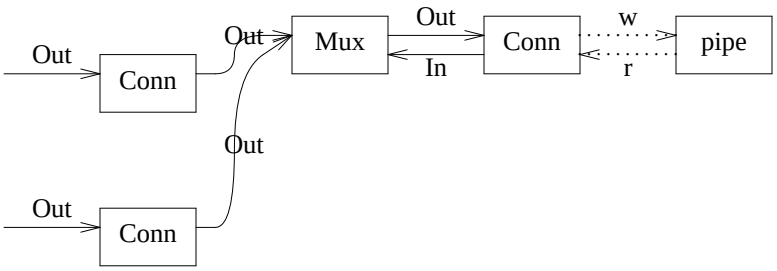

Figure 2: *Multiplexors*

In the figure, the two connections of the left were built by two calls to `mux.Out()`, which returns a `Conn` with an `Out` chan to issue requests. The process using the `Out` channel may issue as many messages as desired and then close the channel, perhaps with an error indication.

If the request depicted below requires a reply, `mux.Rpc()` can be called instead of `mux.Out()` and the result is as shown in the picture:

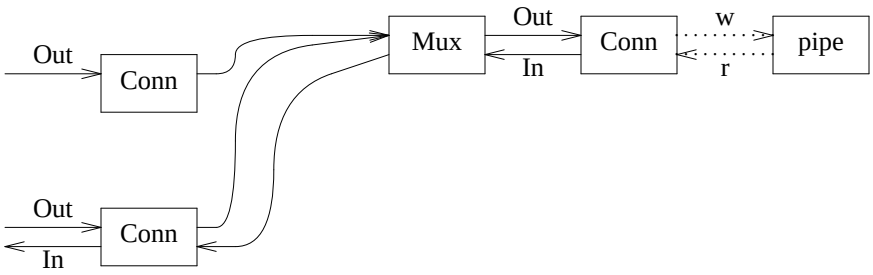

Figure 3: *Multiplexors, ctnd.*

The receiving part of the multiplexor is shown in the next picture:

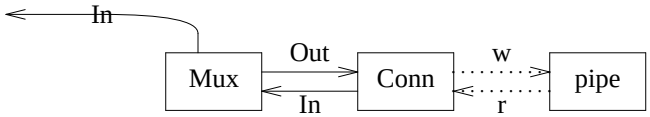

Figure 4: *Multiplexors, ctnd.*

For example, if a process received two requests, one with no reply expected and another with a reply expected, the picture would be:

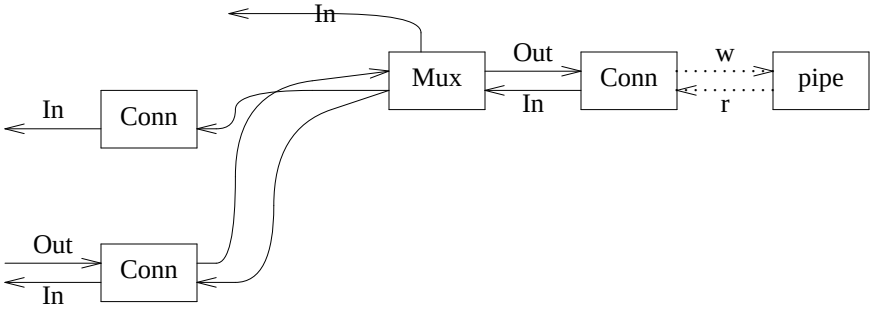

Figure 5: *Multiplexors, ctnd.*

Here, the two connections on the left represent requests that were received through the `In` channel depicted on top of the multiplexor.

The important thing to note is that processes may now issue streams of requests, or replies, through channels and they are relayed to external devices (or from them) as required. The interfaces shown have greatly simplified programming for (networked) system services written for the new system.

Networks And The Dial Service

A popular package for network programming in Clive is the dial service [clive/net/ds(2)]. It is a general-purpose dialing service that provides tools to dial and serve `nchan.Conn` connections, implemented using the tools previously shown.

Most programs take a network address (a string) and then either dial or serve it. The dial service bridges the gap between the address and the `nchan.Conn` connections to/from clients and servers.

Different networks can be registered with the dial service so it knows how to actually dial them and how to serve them. Among the usual ones, there are a couple of interesting networks:

The `pipe` network exists only within the process using it. It relies on pipe connections shown before (not to be confused with Go or UNIX pipes), to provide a network built within the process. For example, this is a complete echo service:

```
// serve the "fifo!*!ftest" network address
reqc, endc, err := ds.Serve("example service", "fifo!*!ftest")
if err != nil {
     dbg.Fatal(err)
}
// receive requests
for req := range reqc {
     if req == nil {
          break
     }
     // serve a request
     defer close(req.Out)
     for msg := range req.In {
          if ok := req.Out <- msg; !ok {
               close(req.In, cerror(req.Out))
               dbg.Warn("send: %v", cerror(req.Out))
               break
          }
     }
     close(req.Out, cerror(req.In))
}
```

Here, we can replace `fifo!*!ftest` with `tcp!*!666` to serve on TCP on all networks, or with `*!*!ftest` to serve on all known networks (provided we defined the port number for the `ftest` service, as needed by networks like TCP. See [addr(3)] for a description of network addresses. New port numbers for services can be defined with `DefSvc` as provided by the dial service.

In the code shown, `Serve` arranges for the process to listen on the network addresses described by the address string (further discussed later), and provides a channel based interface for the server.

The code for a client can be as shown now:

```
cc, err := Dial("*!*!ftest")
if err != nil {
     dbg.Fatal(err)
}
for i := 0; i < 10; i++ {
     cc.Out <- []byte(fmt.Sprintf("<%d>", i))
     reply := <-cc.In
     str := string(reply)
     printf("got %s back\n", str)
}
close(cc.In)
close(cc.Out)
```

`Dial` arranges for the process to dial the named service using the network addresses implied by the address

string and provides a channel based interface.

Name Spaces And Directory Entries.

Name spaces are implemented by prefix mount tables in `nspace` following the `zx.Finder` interface to supply directory entries for `Find` requests, and the `Binder` interface to add and remove entries to the mount table.

A mount table maps from prefixes (absolute paths) to ordered sets of directory entries. There is a mount table per process, which is actually kept as part of the environment. Processes consult the `$NS` environment variable when they need a name space and build one according to the textual representation kept in the variable.

The `Find` request follows the prefixes involved in the search and issues finds to them, returning all the directory entries found. The caller may later dial the file trees responsible for the directory entries as shown early in this paper.

A directory entry is a general table of names and values for attributes, all kept as strings. There are conventions (see [clive/zx(2)] for a description) and some attributes are expected to be there for entries representing files. More on this soon.

But, as far as the name space is concerned, directory entries do not need to refer to ZX files (or to UNIX files).

This means that we can create an empty directory at a existing tree, and then use the name space to mount a series of directory entries at that directory. Such entries may be used like data base entries and can be defined at will. Still, the `Find` request will find them as expected. Only that you will not be able to do file I/O on them.

A binder provides operations to change the name space:

```
type Binder interface {
    // Add an entry at the given path prefix for the given directory entry.
    // If flag is Before or After, previous contents of the path are preserved and
    // the new entry is added before or after them.
    Mount(name string, d zx.Dir, flag Flag) <-chan error

    // Remove entries for the given path prefix (all if d is nil, or those matching
    // if it is not nil).
    Unmount(name string, d zx.Dir) <-chan error
}
```

See [clive/nspace(2)] for more details. But note how this and most other interfaces shown return channels back instead of direct values, so they could be used as promises to stream requests through the network.

A finder implements the primary interface to find directory entries:

```
type Finder interface {
    // Select the file tree to be navigated
    Fsys(name string) <-chan error
    // Navigate the tree starting at path to find files matching the predicate
    // pred. Found entries are sent through the returned channel.
    // Those with errors are decorated with an "err" attribute indicating the error,
    // and they might not match the predicate given to find, if they convey just the error.
    //
    // The server must consider that the path for a file /a/b/c is actually /x/b/c if
    // spref is /a and dpref is /y. That is, spref must be replaced with dpref in paths
    // before evaluating the predicate. This is used to evaluate paths as seen by the users
    // without having to rewrite the predicates at each mount point.
    //
    // The depth for the path given, once walked, starts at depth0 (and not 0), again,
    // to avoid predicate rewriting on mount points.
    //
    Find(path, pred string, spref, dpref string, depth0 int) <-chan Dir
}
```

The name space is the most popular finder, provided by `clive/nspace`

Consider the `Find` request. 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 predicate is considered `true` and indicates all entries rooted at the given path.

But there are more interesting predicates. For example,

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

finds all entries for files (or whatever the named resources might be) matching the given expression for the name, and not looking deeper than 3 levels down from the specified root.

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. Predicates can operate on any attribute defined (or not) in a directory entry. The `clive/zx/pred` package implements and documents them.

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 a file tree, but it can also 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 server providing 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. It may be useful to re-read at this point the examples from the introduction.

Directory Entries

A directory entry is defined in the `clive/zx` package as follows:

```
type Dir map[string]string
    A Dir, or directory entry, identifies a file or a resource in the
    system. It is a set of attribute/value pairs
```

Directory entries are self-describing in many cases, and include the address and resource paths as known by the servers providing them. Thus, programs can operate on streams of `Dirs` and ask each one to perform an operation on the resource it describes. The purpose of very important interfaces in the system, like `zx.Finder` and `zx.Tree` is to operate on `Dir` structures (or rather, on the resources they represent).

When sent through channels, directory entries are packed into byte arrays using a conventional network format.

Some attributes are well-known and are present in many entries:

```
const (
    Size  = "size"  // number of entries in directories, size for files
    Name  = "name"  // file name
    Fpath = "path"  // file path as seen by client
    Addr  = "addr"  // where to reach the server for the directory entry
    Spath = "spath" // path for the resource in the server
    Mtime = "mtime" // mod time since epoch (ns)
    Mode  = "mode"  // permissions (octal, 0777 bits must be unix bits).
    Type  = "type"  // "-": file, "d": dir, ...
    Rm    = "rm"    // whiteout (if set, the file is removed)
    Proto = "proto" // protocols spoken by server
    Err   = "err"   // error for the operation performed at this entry.

    Uid  = "Uid"  // user id or file owner
    Gid  = "Gid"  // group id or file group
    Wuid = "Wuid" // who is guilty if you don’t like the data
    Sum  = "Sum"  // hash for the file data
)
```

The most important ones are perhaps `path` (the path as seen by the user) and `spath` (the path as seen by the server for the resource) and `addr` (the address of the server).

File Trees

File trees are actually resource servers for resources described by directory entries. In fact, there are no files defined. A *file* is just a data type that records which tree serves a directory entry:

```
type File struct {
    T Tree
    D Dir
}
```

The `zx.Tree` and `zx.RWTree` interfaces define what can be expected from a file tree (server). There are smaller interfaces used by many ZX related tools, but these interfaces are easy to grasp and tell the whole story.

Read-only trees provide this interface:

```
type Tree interface {
    // return the tree name
    Name() string

    // Return the directory entry for the file at path.
    Stat(path string) chan Dir

    // Terminate the file tree operation, perhaps with an error.
    Close(e error)

    // Retrieve the contents of the file at path.
    // 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 string with the format
    // produced by Dir.Pack, The end of the file (or dir) is signaled with
    // an empty message.
    // When pred is not "", it is evaluated at path and the get is performed
    // only if the predicate is true. Depth is always considered as 0.
    Get(path string, off, count int64, pred string) <-chan []byte

    // Find directory entries in this tree. See ns.Find.
    Find(rid, pred string, spref, dpref string, depth0 int) <-chan Dir
}
```

As usual, most of these return channels with replies to permit streaming of requests and deferring of error/reply checking until later. The `zx` package has a few tools that operate on a `Tree` and return a direct response.

For example, `zx.Stat` can be used to stat a file and return its directory entry and the error indication, if any:

```
// Helper to issue a Stat call and return the Dir entry now.
func Stat(fs Stater, path string) (Dir, error) {
     dc := fs.Stat(path)
     d := <-dc
     if d == nil {
          return nil, cerror(dc)
     }
     return d, nil
}
```

Its implementation is also an example of how to use the actual `zx.Tree` call. Reply channels, as seen here, convey not just the result of the call, but also any error indication found during the request.

As another example, `zx.GetDir` can be used to read a directory. It calls `Get` and then unpacks each directory entry received.

```
var fs zx.Tree
// ...
ds, err = zx.GetDir(fs, p)
if err != nil {
     return err
}
for _, d := range ds {
     dbg.Warn("got entry for %s", d["path"])
}
```

Refer to the file `clive/zx/util.go` for further tools.

Read-write trees, add these operations:

```
type RWTree interface {
    Tree

    // Update or create a file at the given path with the attributes
    // found in d and the data sent through dc.
    // If d is nil or d["mode"] is not defined, the file is not
    // created if it does not exist, and it is not truncated.
    // If off is < 0 then the new data is appended to the file.
    // Note off<0 makes no sense if d["mode"] is defined.
    // The file mtime and size after the put, or the error is reported
    // through the returned channel.
    // If pred is not "", it is evaluated at the file if it already exists
    // and the put happens only if pred is true. Depth is considered
    // 0.
    Put(path string, d Dir, off int64, dc <-chan []byte, pred string) chan Dir

    // Create a directory at the given path with the attributes found in d.
    Mkdir(path string, d Dir) chan error

    // Move a file or directory from one place to another.
    Move(from, to string) chan error

    // Delete the file or empty directory found at path.
    Remove(path string) chan error

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

    // Update attributes for the file at path with those from d.
    Wstat(path string, d Dir) chan error
}
```

In some cases, we provide a service implementing just part of `zx.Tree` or `zx.RWTree` and providing only a few operations. The tiny interfaces (one per operation) in `zx` and the functions `zx.TreeFor` and `zx.RWTreeFor` wrap an interface providing default (failing) methods for all not implemented by the argument.

Code that requires only part of the interface for a tree may rely on one of the small interfaces also declared in `zx` `zx.Tree` and `zx.RWTree` are easier to understand what has to be implemented/provided by a file tree.