ZX: A distributed file system for high latency networks

Francisco J. Ballesteros, Gorka Guardiola, Enrique Soriano

ABSTRACT

File access today has network latency as one of the main problems. Some systems rely on aggressive caching to address such problem, others rely on batching. In this paper we present ZX, a distributed file system and protocol designed with latency in mind, for the Clive OS. It can use caching, but it is not centered on caching or batching to address latency issues. ZX relies on a novel file system interface, which includes find requests, and on streaming RPCs to work well under high latency conditions. With round trip times on the order of 50ms or 100ms it can still be used to access a central file server to compile shared sources, run binaries, edit documents, and similar workloads. It can be used on UNIX using a FUSE adaptor while permitting native ZX speakers to run faster.

Keywords: Distributed File System, Operating System, Cloud Computing.

1. Introduction

ZX is the file system for a new OS, Clive [1], built to permit the construction of efficient Cloud services. It supports file access through LAN and WAN links, which exhibit a wide range of latencies. Ubiquitous access to the internet and mobile devices make this a very common scenario, raising latency as one of the main issues for remote file access.

There are many cases today where the latency can go over 100 ms of RTT, the common one being an overseas connection. For instance, the RTT from our University at Madrid to Bell Labs in New Jersey on a good day is about 110 ms over a cabled network. The straight distance from one point to the other is limited by the speed of light in the vacuum which is around 40 ms of RTT. This is the bare minimum, but of course, the cables do not follow the straight line and are not always fiber. Even in fiber, the speed of light is about 1/3 lower than in the vacuum. Also, routers (perhaps congested), switches, and other intermediate machines have to be taken in account. This latency can be worse over a wireless or 4G link when there is enough noise, or the signal strength is small.

Many distributed systems, for example [2], ignored this issue because they were designed for local or metropolitan area networks. NFS [3] was designed initially as a file protocol for LANs and later evolved to consider network latency and perform better with worse latencies.

Using a central file server is good for coherency, regarding file access, but as latency increases, remote call round trips quickly add up and degrade the service. With high enough latencies (eg., RTTs of 50ms or more), the file service may be slow enough to make it inconvenient for actual file access. As a result, other strategies for sharing files, like those using loosely coherent replicas that rely on a synchronization protocol have become popular. For example, using NFS through a WAN might work but it would be slow enough to make it inconvenient. Therefore, most users would consider the service as unavailable because of its (poor)

1. This work has been funded in part by the CAM project S2013/ICE-2894 cofounded by FSE and FEDER, and by the Spanish MINECO project TIN2013-47030-P.

performance and switch to a file replica tool, even in those cases when a central file server might be preferred.

ZX is an effort to let using a central file server work well under bad latencies to provide better coherency for access to remote files. Although it is not designed for disconnected operation, it is a good choice for cases when access to the network is available, perhaps through bad links exhibiting 100ms of RTT. With such latencies, it is still practical to use a central ZX file server for software development, accessing user and system files, running binaries, writing documents, etc. The results shown later, and our own experience using this system on bad links, support this claim.

One reason why many file systems have problems in the scenario considered is because they were designed following the UNIX file system interface and its system calls. This interface was perfect for the 70s, but today, considering the network, better interfaces might be desirable. The reason is that such interface implies many roundtrips from the application to the file server. As others have noted [4], a better interface is required for WANs. ZX departs from the venerable UNIX interface for file access, and relies on FUSE adaptors to let UNIX applications work.

Another important reason is that many file systems rely on RPCs to provide their services, and latencies add up quickly when using RPCs for remote access. The synchronization implied by the conventional RPC mechanism leads to the addition of round trip times and, when latencies go above 10ms and become closer to 100ms, makes it unpractical to use the resulting protocol.

Considering together both the UNIX interface and an RPC implementation it is not a surprise that many users rely on alternative access methods like Dropbox or Google Drive.

ZX exploits three features to meet its design goals:

- The interface used to find files of interest in the server(s) is separated from the interface providing access to file data. That is, it is feasible to let a server find files (or rather, their directory entries) and stream those to the client.

- All requests have an interface suitable for streaming data both to the server and from the file server.

- Calls for file access do not return the result directly, but return a channel that permits retrieving it at a later point in time.

Together, these permit programming and using applications that perform file access in a convenient way, yet they still work well under bad latency with good coherency. Of course, network outages forbid file access and, introducing a cache to permit access during network outages harms coherency. The user must choose between file availability during disconnected operation and coherent file access.

In ZX, a request is a stream of data that is replied with another stream of data. Procedure calls for file requests accept input channels when they require data to be sent to the server, and always return reply channels. A reply channel may be used right after the call, or at any point in the future, to retrieve both the status of the request and any data or error indication from the server, resulting from that request. Programming with this kind of interface is as easy as programming with the UNIX-style interface for file access, but the underlying streaming helps with latency by improving round-trip times as seen by the user.

A central request in ZX, *find*, permits the client to issue a predicate to find files. As a result, many roundtrips to the server are avoided to locate files of interest. Most applications access files and directories to determine if they meet a desired property before using them. For example, a compiler might just read some directories for included packages to locate source or object files, a word processor might read multiple directories to locate image files for inclusion on a document, etc. In all these cases, a single *find* request

can stream desired directory entries back to the client in a single request.

With bad latencies, this feature alone makes the difference between being able to use the file system or not (i.e., or going for an alternative that sacrifices coherency). The drawback of relying on *find* is that interruption of ongoing *find* requests wastes both server computing resources and network bandwidth, because the server will still be processing the request for a while until it notices that the request has been interrupted. This is the price ZX pays to tolerate bad latencies in a better way.

ZX is also interesting regarding its metadata handling. To be system neutral, a directory entry for a file (or its metadata record) is just a set of name/value attribute pairs. The semantics for them are left out of the protocol in many cases. Of course, attributes like *size* and *name* have the expected semantics, but any system (or user) is free to define its own attributes and to apply the desired semantics to them.

Other systems like iRODS [5] and POSIX extended attributes [6] permit arbitrary name/value attributes. But unlike them, ZX metadata follows the convention that attributes with names starting with uppercase are not to persist in the file system storage. This makes it easy for programs to decorate directory entries by adding their own attributes for their own purposes. In a way, this permits defining "temporary" attributes, which is useful on its own. Caches may add checksums and cache time stamps, errors for file access may be noted in directory entries sent to other programs, and so on.

In what follows we describe the ZX file system interface, the protocol underlying it (also called ZX), and several tools using it. We include a description of a ZX cache and a FUSE adaptor for UNIX that can be used to operate with disconnections, and to run UNIX programs that are not native ZX speakers. We describe also how native ZX tools exploit the protocol to perform better. Finally we describe related work and present quantitative evaluation results.

2. The ZX file system

Both ZX and Clive are written using a modified Go [7] compiler. The Go language follows the style of Communicating Sequential Processes [8], or CSP. To be precise, Go follows a Bell Labs concurrent style, where processes exchange data through channels.

Channels are first-class citizens in Go. They may be buffered or not. Sends proceed while there is buffer space in the channel, and block waiting for a receiver in other case. Receives take messages from the buffer space (if any) or block waiting for a sender. A Go channel may be closed to notify receivers that no more data will be sent. Go uses "`<-`" as both the receive and send operator, depending on which side of the channel it is written; and includes a `range` construct that loops receiving from a channel until "EOF".

The modified Go compiler [9] used by Clive permits `close` to be used at any point, in a way that signals an error (given to close) to both senders and receivers of the channel. Resulting channels work in very much the same way pipes work in UNIX, which is a departure from standard Go (standard Go would cause a panic instead). In our compiler, the error supplied to `close` may be retrieved using a new `cerror` primitive.

For example, this code excerpt is the idiom in Clive for receiving data from a channel (`inc`) and sending it through another channel (`outc`):

```
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))
```

The loop receives messages and the conditional checks out if the modified data could be sent to the output channel. Upon errors on a channel, the other channel is closed with the error diagnostic and the process ceases its processing.

This scheme of operation inspires the interface for all ZX operations. A ZX RPC is actually a stream of input data for the RPC, sent through a channel, and a stream of output data sent through a reply channel. The reply data includes the overall status for the request, sent by closing the output channel. When the file server decides to abort a request, the input channel is closed, and any sender would get an error when trying to send anything as part of the request. The output channel is also closed upon errors.

Of course, the sender might have sent quite a bit of data before it notices that the request was aborted by the server, but the wasted bandwidth is part of the price for avoiding extra latency in the common case when the operation is performed without errors.

2.1. Finders and file trees

Unlike most other file systems, ZX splits its interface into two different entities:

- A *finder* interface, used to find directory entries.
- A *file tree* interface, used to operate on files.

Before describing them, an example may clarify the implications of the resulting design. In Clive, a user may list all directories under `/sys/src` using

```
> lf /sys/src,d
```

and may list all Go source files under `/sys/src` using

```
> lf /sys/src,~*.go
```

Albeit using a cryptic syntax, "`/sys/src,d`" is an expression to select files of interest, as is "`/sys/src,~*.go`". Such expressions combine both a file name ("`/sys/src`") and a predicate ("`d`" and "`~*.go`"). The predicates used here are abbreviated forms for "`type=d`" (i.e., file type must be a directory) and "`name~*.go`" (i.e., file name matches the given globbing expression, that is, file name suffix is "`.go`").

The Clive shell issues a single *find* request with the given file name and predicate, and receives a stream of matching directory entries (or errors) from the server. The code looks like this:

```
dirc := ns.Find("/sys/src", "type=d", "/", "/", 0)
for dir := range dirc {
    if long {
        cmd.Printf("%s\n", dir.Long())    // long listing
    } else {
        cmd.Printf("%s\n", dir["path"])    // print its path
    }
}
if err := cerror(dirc); err != nil {
    cmd.Warn("%s", err);
}
```

A ZX *Finder* is anyone implementing this interface:

```
Find(path, pred string, spref, dpref string, depth0 int) <-chan Dir
```

This request navigates (in the server) the tree at *path* to locate directory entries matching the predicate *pred* and streams them through the output channel (returned). Remaining arguments indicate a path prefix that has to be replaced in replies with another prefix, also given, and the depth for the path in the file tree (name space) as seen by the client. This permits the client to rearrange its name space (similar to what a UNIX or Plan 9 mount table does) yet permits the server to evaluate the predicate using the paths and depths as seen by the client.

As depicted in figure 1, there is a single "RPC" issued to the server, and all round-trips that would be needed to locate each file are now local procedure calls within the server. Replies to directory reads in protocols like NFS may look similar, but they are rather different from the ZX interaction. For example, for a single directory, NFS may also stream directory entries back to the client. But note that *find* may traverse a full file hierarchy and may filter entries of interest using its predicate.

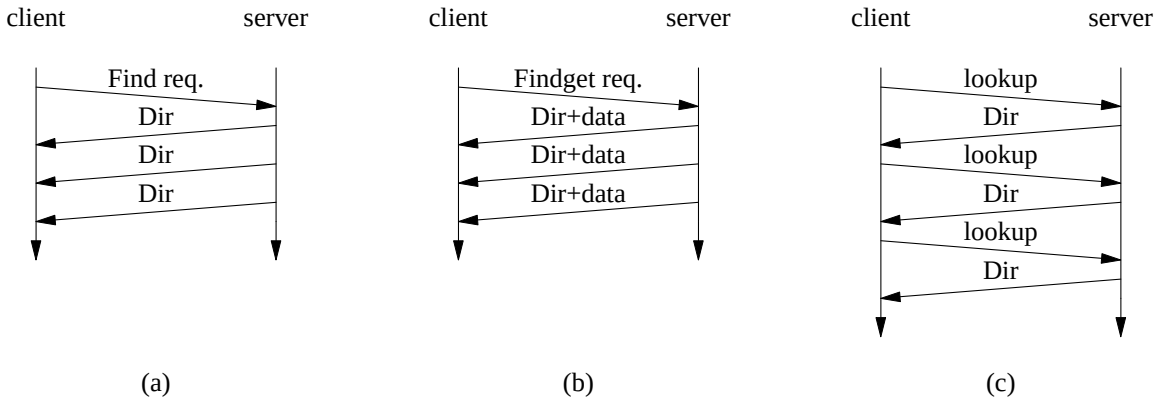

Figure 1: *Sending multiple messages per request and reply leverages streaming and minimizes the effect of network latency in execution times. From left to right: (a) Find call, (b) Findget call, (c) Interaction using UNIX-style RPCs (time flows down).*

Also, note the implication of making *find* return a channel for addressing latency problems. The client program might issue multiple find requests and later receive their replies and/or combine and forward them to someone else. This is depicted in figure 2, where the difference in round-trips with respect to a batching protocol can be seen. The channel interface permits the client code to issue multiple requests without

having to wait for the first one to proceed with the second one. It must be noted that a batching protocol might also be able to issue multiple concurrent batched requests (if the protocol permits doing so), but even in this case, if requests are finding directory entries, multiple sequential batches may be required.

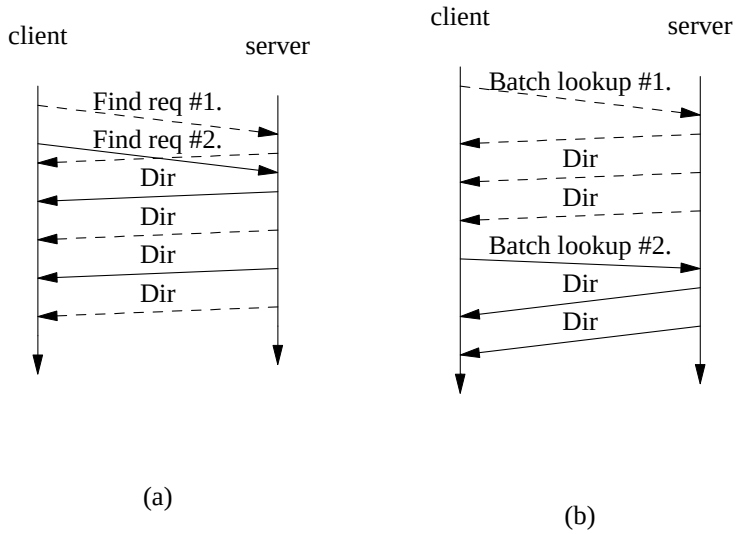

Figure 2: *In ZX (a) two find requests issued may proceed concurrently and their reply channels may be processed later. In (b) a batching protocol must wait for (batched) replies when traversing a tree to find files because further requests may depend on previous findings.*

A related operation, *findget*, operates in the same way but returns both metadata and data for matching files. The result retrieved from the output channel is a series of directory entries for each matching file, with file data following each directory entry in the reply stream as shown in figure 1.

```
FindGet(path, pred string,
    spref, dpref string, depth0 int) <-chan interface{}
```

The channel returned conveys two different data types (directory entries and raw byte arrays) and thus we use the Go type `interface{}` which is just any data type.

As an example, to read all Go source files and search for a string in them, we may change the example above to issue a single *findget* and then operate on the data received. Again, there is a single round-trip to the server and we may exploit streaming to aid with latency. Figure 1(b) shows the effect. With a conventional UNIX-style interaction, further RPCs must still be added to figure 1(c) to retrieve file data, although batching protocols (eg., later versions of NFS) may still retrieve data and metadata in a single batched request.

This request exists because ZX is very aggressive regarding avoiding round-trips to the server. It avoids the need to call *get* for all files located by a previous *find* request.

The separation between *finders* and *file trees* is very important for addressing high latencies. The next example illustrates this point. It removes all object files in our source tree. In the example, we defer checking error status for file removals until all requests have been sent. Also, the stream of matching directory entries is being sent to the program from the server(s) while it issues remove requests for them. It is likely that most, perhaps all, of the replies for remove requests arrive before the code actually checks for errors.

```
dirc := ns.Find("/sys/src", "~*.o", "/", "/", 0)
errors := []chan error{}    // list of error channels
for dir := range dirc {
    // get a handle for the dir's tree
    wt, err := zx.RWDirTree(dir)
    if err != nil {
        cmd.Warn("%s: tree: %s", dir["path"], err)
        continue
    }
    errc := wt.Remove(dir["path"])
    errors = append(errors, ec)
}
for _, errc := range errors {
    if err := <-errc; err != nil {
        cmd.Warn("%s", err)
    }
}
```

Figure 3 compares the series of round-trips required to perform the same operations using *find*, using a protocol without *find* but with batching capabilities, and using a UNIX-like interface and file system protocol. The interaction shown in 3(b) is a mixture of batching and streaming protocols. A batching protocol would have a single batched reply for each batched request and a streaming protocol would issue a stream of replies (as shown). In any case, in practice, both batched/streamed requests and UNIX-style requests would require more round trips than shown in the figure, because *find* knows how to traverse a file hierarchy but most file lookups in other protocols do not.

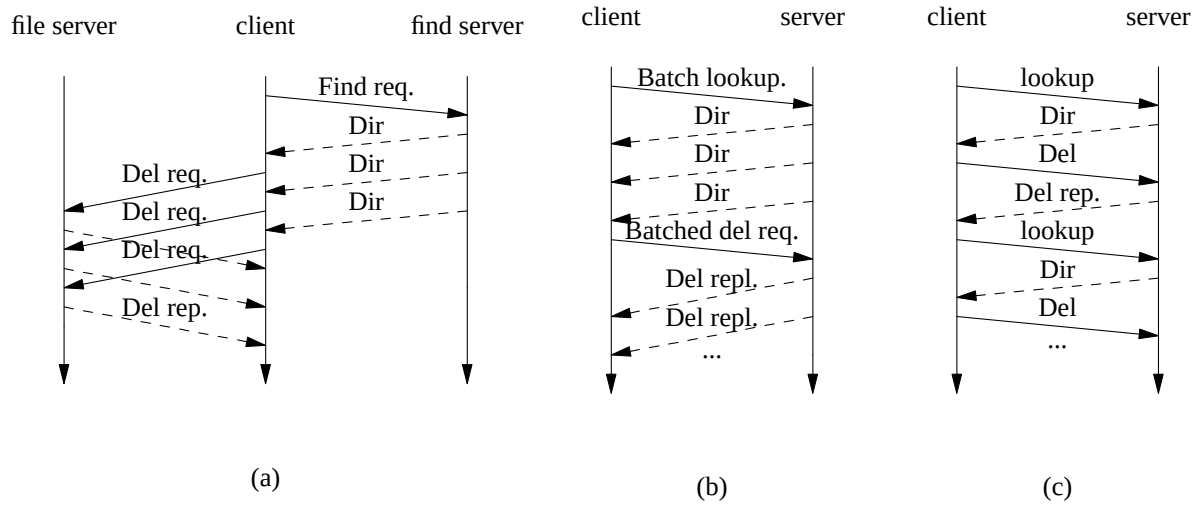

Figure 3: *Issuing a series of remove requests for files (a) with find, (b) with batched lookup and remove requests, and (c) with classic UNIX-style requests.*

Going back to the example, each directory entry is self-describing, and includes as an attribute the address for the file in the network (i.e., the address of its server and the path in the server for the file). That is how `RWDirTree` can return the handle for each one. If a connection to a server is available, it is used; otherwise the server is dialed and a new connection is set up for this and further directory entries on such server.

The interface for a ZX *Tree* provides these requests (or a subset):

```
Stat(path string) <-chan Dir
Get(path string, off, count int64) <-chan []byte
Put(path string, d Dir, off int64, dc <-chan []byte) <-chan Dir
Move(from, to string) <-chan error
Remove(path string) <-chan error
RemoveAll(path string) <-chan error
Wstat(path string, d Dir) <-chan Dir
```

Return channels can be used very much like promises or futures [10]. Furthermore, those returning a single value are buffered and they might be ignored when the value in the reply is not of interest. All operations in the ZX interfaces are designed along the same lines.

Stat returns a directory entry for the given path. *Move*, *remove*, and *removeall* are self explanatory. *Wstat* can be used to update the attributes supplied in the given directory entry, or to remove them if the value supplied is a null value (the resulting directory entry is returned).

Get and *put* requests are more interesting in that they can stream data from the server (or to the server), yet they are close enough to traditional read/write semantics to make it easy to write adaptors for using other file interfaces like FUSE.

Get returns a channel to retrieve data from the file starting at a given offset for a maximum of the indicated number of bytes.

Put is used to store data in the file starting at a given offset. If a directory entry is supplied, it is used to create and/or to update the file metadata at the same time. A file creation is requested by specifying the file type in the directory entry. Therefore, *put* can save several roundtrips because all the data (both data and metadata) can be sent along in a single request. Permissions are set before actually placing data in the file at the start of the request, and the final file modification time is set at the end of the operation using either the given value for such attribute or the current time. The resulting set of attributes after *put* completion is returned. Thus, caches might exploit the resulting size, modification time, and perhaps SHA1 for file contents if they are provided by the file system implementation; editors might use them to detect if a file was externally modified before saving its contents; etc. This saves extra `stat` requests in these and other cases, which further aids with latency.

There are no further calls; the ones shown suffice. For example, *wstat* can be used to update the `size` attribute and perform a truncation or a resize of the file. Thus, there is no *truncate* call. File creation happens whenever a file does not exist and *put* is called with a directory entry providing a file type. Instead of *mkdir*, a *put* specifying "directory" (or "`d`") as the file type is used to create a new directory. Note that resizing might be performed also as part of a *put*: creating a file with a single hole of 1Gbyte and then writing a portion of data within the file can be performed with a single request. In the same way, updating both file permissions and file data can be performed with a single request.

2.2. The ZX file system protocol

The ZX protocol follows from the design of the operations shown before. Its messages are shown in table 1.

All messages start with *size* and *tag* fields not shown in the table. They are little-endian 8-byte numbers used to indicate the total size of the message (not counting the size field) and to match replies with requests. Tags used in the protocol identify the channel for each message exchanged. They are used to bridge Go channels through the network; i.e., to multiplex the network connection among multiple channels or requests. This permits streaming of data through a channel instead of, for example, using multiple *write* RPCs like UNIX would do.

The *type* field is a single byte indicating the message type. Remaining fields match the arguments of the

	Type	Fields
put	Tput	path off dir
get	Tget	path off count
move	Tmove	path topath
remove	Tremove	path
rmall	Trmall	path
wstat	Twstat	path dir
find	Tfind	path pred spref dpref depth
findget	Tfindget	path pred spref dpref depth
fsys	Tfsys	name

Table 1: *Requests in the ZX protocol. Each request is sent before any raw data it requires. The replies are directory entries, errors, or raw data sent as a reply stream for each request.*

corresponding ZX call. Fields *path*, *topath*, *pred*, *spref*, and *dpref* are strings, encoded by sending their length and then their UTF-8 text. Fields *off*, *count*, and *depth* are 8-byte little-endian integers. Fields *dir* are directory entries encoded by sending the number of attribute/value pairs and the sequence of attribute and value pairs, encoded as strings.

As shown, there are *no* data fields in any of the messages. If there is data flowing to (or from) the server due to a request, the data follows the request in the channel used to issue the request (or in its reply channel).

There is another message, not shown, used to implement flow control. It is sent when half the buffer space at a receiver has been consumed, to signal the sender channel at the other end of the network connection and let it continue streaming. Flow control as implemented is naive, but prevents a request streaming enough data for an unresponsive reader to collapse I/O for other requests. Nevertheless, any (better) flow control scheme may be adopted if necessary.

Requests used to read and write a file require further examples. This may be the code of the client program used to read a file:

```
datachan := ns.Get("/foo", 0, zx.All)
for data := range datachan {
    if err := Process(data); err != nil {
        close(datachan, err)    // don't want more
        break
    }
}
status = cerror(datachan)
```

Here, a `Tget` message is sent through the network. No further data is sent through the request channel. As a reply, file data is streamed back and received by the client. Should there be an error (eg., a "permission denied"), the server would close its reply channel with such an error. In this case, no data is received by the client and the call to *cerror* yields the error message describing the problem (both the capability of closing an input channel and *cerror* were added to our Go compiler and runtime and are not part of standard Go).

This is the code used to update (or create) a file:

```
datachan := make(chan []byte, 1)
datachan <- filedata
close(datachan)
dirchan := ns.Put("/bar, Dir{"type":"-"}, 0, datachan)
filedir := <-dirchan
status := cerror(dirchan)
```

Here, we create a channel with buffering, send a single bunch of data through it, and close it to signal that there is no further data. Then, the call to `Put` issues a `Tput` message through the request channel and streams data from `datachan` through the same channel. In this request, the reply message is either an error or a directory entry sent through the reply channel.

The last two lines of the code shown receive the directory entry for the updated file and check the error status for the entire operation. The directory entry would be `nil` when errors happen.

When the server receives the `Tput` request it processes the request and, perhaps, closes both the request and reply channels with an error indication if there is an error. In this case, the ongoing data to the server is discarded when it arrives.

When the server refuses to perform the `Put` and the client program sends the data *after* calling `Put`, the sends fail and report the error as soon as the error is received by the client side. The client then stops sending data and handles the error. The resulting code is exactly as shown in the first example of section 2.

Usually, the request has no errors and proceeds normally. In this case, the client is already streaming data while the server starts processing the request. ZX is optimistic in this respect, to decrease the latency of its normal operation.

Streaming in ZX has implications for consistency semantics. Each request *message* is meant to be processed atomically with respect to others involving the same file. But it is important to state that it is the *message* and not the request stream the one with atomicity. For example, in a *put* request, when `Tput` reaches the server, the file server checks the file permissions and request parameters, and perhaps creates or truncates the file as dictated by the arguments in the request. Should concurrent puts happen on the same file, their `Tput` messages are still processed atomically one after another. But this does not apply to the data streamed after each `Tput` message. Such data is written at an offset implied by its `Tput` request and by previous data streamed in such request. Data written by two concurrent *put* streams would interleave their writes, as UNIX would do. This departs from systems like Coda [11], which rely on session semantics were full files are updated at a time.

2.3. Caching

In most cases, direct operation between a ZX client and its server(s) suffices to perform the work at hand. However, caching may be useful to avoid retrieval of data already sent to a client (at the client) and to avoid reading data from disk (at the server). Also, a cache might support disconnected operation by accepting ZX requests that operate on the cached data while disconnected. In this case, we would fall into the loosely synchronized replicas model of operation (this is out of the scope of this paper and it does not differ from what others have done before).

There have been different implementations of caches for ZX in Clive. Here we describe those of the first edition of the system, but they are illustrative of how the file system and its protocol cooperate for caching, which is the point made in this section. We wrote four combinations of memory/disk and write-through/write-behind caches for ZX.

Due to the design of the ZX interface, a write-through cache intercepts a request stream and forwards it to the server. Other systems, like Coda [11], use session semantics because issuing one RPC per write request is unbearable. Systems like NFS [3] rely on client-side caching and/or batching for multiple requests.

However, ZX may just continue streaming data while preserving the semantics of its interface. This avoids the need for an extra RPC to the server each time data is being put into the cache, i.e., each time *write* is called. This benefit comes from the new interface used for updating files; it is used instead of the old open/write/close interface.

A delayed-write cache may also benefit from ZX. It can stream data to the server (as a client would do), instead of issuing multiple write requests. The speed up would not be perceived by the end user, but it reduces the whole system load and it is more efficient because the cache can leverage streaming capabilities and decrease the number of requests issued to update its entries and to forward updated files (and file attributes) to the server. Of course, in this case consistency departs further from UNIX, although the client buffer cache in most UNIX kernels is actually a delayed-write caching and suffers the same effect.

The cache may issue *find* requests to the server to serve client *find* requests and, if the intercepted directory entries show that the cached data is current, serve the data itself. Doing it this way provides better coherency despite caching because the server has been reached to decide if cached files are up to date or not.

2.4. ZX on UNIX

Two programs cooperate to serve ZX file systems in Clive for UNIX. *Xzx* exports a ZX file system to the network. *Zxfuse* is a client for ZX that serves a FUSE file system for UNIX.

The FUSE driver includes a cache, because the UNIX kernel issues many requests as implied by its UNIX file interface. It can also operate without caching, should the user ask so. Any of the caches mentioned in the previous section may be used, because they present a ZX interface and *zxfuse* uses that interface to talk to them.

Being a FUSE driver, its interface to UNIX is similar to the one provided at the filesystem level. That is, there are operations for i-nodes (or rather, v-nodes) and the UNIX kernel issues multiple requests per system call. Every time UNIX wants to reach the i-node attributes it would issue a *getattr* (or *stat*) request. Opening a file usually implies a series of *getattr*, *opendir*, and *readdir* requests to traverse the path, and then a call to *open* or *create*. If the file is being read, a series of *read* requests would follow.

When used without any caching, this implies a series of ZX *stat* and *get* requests to retrieve the information from the server. The directory entry cache kept by the UNIX kernel saves some of the required calls, but still, caching is suggested. With or without caching, for sequential reads, a single *get* request still streams data from the server (in the hope that further UNIX reads would continue reading). In the same way, for sequential writes, a single *put* request suffices. When random access is used (without caching), one *put* request per write must be issued by the ZX FUSE driver, and the same happens for random-access reads (without caching enabled).

In the case of the write-through cache, accessing a directory entry makes the FUSE driver issue a *get* to read the entire directory (in the hope that sibling directory entries might be also used). Such entries expire in a short time (1ms), but that suffices to let the UNIX kernel issue multiple *getattr* requests against the cached entries. Opening a file for reading, writing, or both, implies a ZX *get*, a ZX *put*, or both. In the case of the write-through cache, sequential writes share a single *put*, and random-access writes rely on a *put* per write (because each one must operate on its own file position). Sequential reads operate with a single *get*, as do random access reads (because the entire file is retrieved with a single *get* for further use). Once a file is cached, directory entries retrieved from the server are used to invalidate old cached data.

When no cache is used, consistency is similar to that of UNIX. It is similar and not exactly the same, because if reads are sequential, a single *get* streams data from the server and it might happen that data has been streamed before a concurrent write happen.

When using a cache, consistency depends on the cache used. Write behavior is similar to that of UNIX (because each write message is processed atomically at the server and it is seen by the cache). Other clients of the same ZX FUSE driver are still under the typical UNIX coherency model, but they might miss file updates made by other machines after the file was open but before the file data is invalidated.

Zxfuse does not use *Direct IO* but for control files (described later), which are synthetic files that permit the user to issue control requests for the file system, in the spirit of Plan 9's control files [12].

The file tree served is re-exported through a local UNIX connection, for native ZX speakers running at the client machine. As a result, we leverage existing UNIX tools and programs (using the FUSE service) while we permit the execution of native Clive tools (speaking ZX directly without going through FUSE). The user may therefore get the best of both worlds.

The evaluation section provides more information on the behavior of the FUSE driver. An interesting point is that *SSHFS* [13] does not work well enough to run the `Go install` command and sometimes it complains about a corrupt object or binary, which means that our *SSHFS* is not providing the expected UNIX semantics. However, such command runs fine when using *Zxfuse*, which indicates that it is close enough to UNIX semantics to let it work. That is not to say that *SSHFS* is not a fine tool (which it is); but to say that ZX can be adapted correctly for UNIX interfaces and legacy programs.

A Clive Go package, *zxux*, provides a ZX interface for existing UNIX files. This can be used to export UNIX file trees as ZX file trees. Attributes are kept in a, per-directory, "`.zx`" file so that the interface presented to users include directory entries attributes as expected in Clive. Note that some systems do not implement extended attributes and using them would harm the portability of the tool.

2.5. Control interfaces and file system stacks

Most ZX file servers provide a `/Ctl` file, which is not an actual file stored with the rest of the files. It is a control interface to operate on the file system. Users may read `/Ctl` to inquiry the status of the system and may write textual commands into it to issue control requests. Tools like the UNIX `grep` command (or Clive's `gr`) may be used to see who is using the file system:

```
> grep user /zx/Ctl
user    rfs:193.147.15.27:65348 nemo as nemo on 2015-03-26 11:40 CET
user    rfs:193.147.15.27:65349 nemo as nemo on 2015-03-26 11:40 CET
>
```

In the same way, we may see the status of the debug flags and change them using `echo`:

```
> grep debug /zx/Ctl
fdebug off
vdebug off
debug on
ldebug off
rdebug off
> echo vdebug on > /zx/Ctl
>
```

There are no extra file system calls required to operate on ZX file servers, `/Ctl` control requests suffice. This is not new, the same idea has been used for decades in Plan 9. But there is an important difference when using nested file systems: ZX can nest control files. To the best of our knowledge, ZX is the first file system using nested control files.

Consider, for example, a ZX FUSE server at the client machine which relies on a ZX cache at the client, which speaks to a remote ZX server, which may be a ZX cache for a on-disk ZX file server. The scenario is depicted in figure 4.

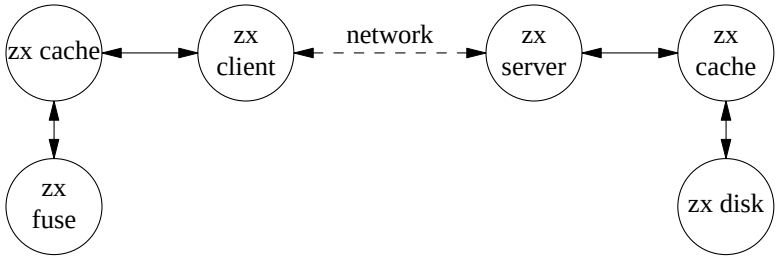

Figure 4: *A stack of ZX file systems to provide cached FUSE access to a memory-cached server for a on-disk file system.*

In this case, each ZX server in the stack reports as the contents for `/Ctl` both its contents and those of the next server in the stack. Reading `/Ctl` reports the status for all the systems in the stack, and we may see the status for any of the file systems involved. This includes debug flags, the list of users logged in, usage statistics, etc.

When a control request starting with "`pass`" is written to `/Ctl`, the server strips the "`pass`" prefix and issues the resulting update to the next server in the stack. So,

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

switches on the `debug` flag for the third file system down the stack.

On servers using caches, writing `sync` into the control file synchronizes any write-behind cache. Unlike most other requests, this one is passed to all file systems in the stack, synchronizing all of them.

2.6. Replicas and history

Clive takes a toolbox approach and provides separate programs to record the history of changes in ZX file trees and to replicate them. This is not new, but it makes the point that it is easy to include such features for ZX file systems.

The *dump* program scans a ZX file tree and archives it at a separate location. It de-duplicates files by computing the SHA-1 for each file (or directory) and storing it once (even if it is found at different places in the file tree). As Plan 9 does, we run *dump* once per day to record the history for our main development tree. A related command, *hist*, searches the dump to compute file differences or copy files back from the past.

The *repl* tool synchronizes ZX trees. Its operation is similar to any other file replication tool and is not relevant for this paper. However, it is interesting to note that *repl* issues a single *find* request to retrieve metadata for the synchronized trees. It then process the output stream of metadata for the tree to detect changes that might have to be propagated to the replica at hand.

When files must be updated, ZX streaming interfaces aid in operating through poor network links.

2.7. Failures

Errors in server operations are reported by replying with error indications to the client, as part of the reply stream. Failures in the network or crashes of the server machine are a different issue. In general, errors in the network connection are handled by the network protocol stack and reported to the implementation of

the ZX client (or server). Our implementation for ZX takes a network connection (usually TCP) and uses it as-is. Handler functions are used to serve and dial ZX connections at given network addresses, and they may optionally activate keepalives. Most of our ZX clients activate keepalives, which means that after some time upon failures, the network connection is broken by the system and an I/O error is reported. At this point, the I/O error is forwarded using reply channels of ongoing operations (i.e. they terminate with failure).

The implementations for ZX client caches use timeouts to speed up the detection of broken connections, in very much the same way other protocols (eg., the kernel implementation for NFS) use them. However, one of the caches implemented for the client side tolerates disconnections and permits using the cached files while disconnected, attempting to reconnect from time to time. Once reconnected, it tries to synchronize its changes to the server (the newest file versions are kept, and there is no conflict resolution in this implementation).

Should the underlying network connection decide to block instead of reporting an error, the requests made by ZX to send or receive data would simply block instead of failing, and the requests for the channels going through such connection would block as well.

As a result, the performance of the protocol in the presence of failures heavily depends on how the rest of the code configures the underlying network connection and also on the usage of timeouts by both the network implementation and the code calling a ZX operation. In few words, an operation might take as much time as required to reach a time out.

All this is to say that I/O errors and server failures (crashes) are handled like in most other file protocols, by reporting them to the caller when they happen. When a program uses ZX, it is free to decide whether to use timeouts or not, as well as what to do when errors are reported.

3. Pitfalls and drawbacks

We made many mistakes while designing and developing ZX. An important one was making *put* atomic (at early versions of the system). This kept files locked while data was being sent from the client. While this may be reasonable for UNIX during writes, it is not for the entire update of a file. A broken client may hold the lock on the file for too long (or forever). In the present implementation only the initial processing of *put* requests is atomic. Once the data update phase is reached, the lock is released. It was nice to be able to put and get files atomically, but it is too dangerous.

A related design failure was considering session semantics, like in [11]. In a discarded prototype with session semantics files were synchronized by client caches when all the data was received. This is fine when different clients do not interact by sharing files. However, updates made by one client are not seen by other clients until the entire *put* completes. Some legacy programs, including editors, kept files open and retained changes made. For example, while editing, it was not feasible in some cases to use other client machines to run programs using the files open in the editor. We discarded such design in favor of the one presented here.

Another pitfall, still present in the current implementation, is error reporting for FUSE. Using the expected error codes makes some of our UNIX systems die or disconnect the FUSE driver. For now, we report most errors as permission problems, and print the actual diagnostic on the console of the driver.

The implementation of the *move* request was a common source of bugs, as were caches. Currently, we flush all caches in the stack before issuing a *move* and then let the end server perform the operation. Any cache up the stack will notice later and update its contents when any involved files (source or destination) are used.

Another problem is actually a consequence of the design for ZX. It works well while far away from the servers and thus it is often the case that there is no copy of the files at the laptops used as terminals. When

the network breaks or is unavailable, we no longer can access our files. But this is the drawback of avoiding a loosely coupled cache for file access. This is also the case when the servers get down due to a power failure at the building or any similar problem. This problem may be addressed to some extent by hosting the server in the cloud, although we keep our own servers for now.

To address disconnections, we recently added disconnected operation to one of the ZX caches. While connected, it operates as described in this paper. Upon network outages, operation continues with the cached data after issuing a warning to the user. Later, upon reconnection, the cache tries to synchronize to the server and does what it can (as many other cached file systems described in the related work would do). Of course, there is no coherency when operating in this way.

The *find* operation might be seen as a "batch" mechanism, in that it replaces multiple RPCs that would be required otherwise. Unlike generic batching systems, ZX is not able to batch unrelated RPCs into a single one. However, this permits better (and simpler) error handling and *find* suffices in most of the cases to avoid the need for batching, combined with the streaming capabilities of the RPCs used by ZX.

Because the server may stream data to the client even for file trees (consider *findget*), it the client decides to abort a request the server may still perform unwanted work for a while, unlike protocols that operate in lock-step issuing one request at a time. As stated before, this is a price we pay for better latency.

Last but not least, the departure from the UNIX API makes it necessary to use adaptors (such as FUSE) to use ZX for legacy applications, and this comes with a significant overhead when compared to native, local, file systems. The overhead comes from extra time required to call the user-lever FUSE driver from the UNIX kernel. For networked file systems, the overhead pays off as soon as latency increases, which is the case addressed by ZX. For a well-connected local area network NFS might be a better choice.

4. Related work

There are uncountable systems and papers published on distributed file systems since decades ago. Due to space constraints, we address here the most significant ones regarding the work presented in this paper. After apologizing for not citing others, we have to say that ZX is a departure from them in that it leverages the CSP style of programming and introduces a *find* request for providing a distributed file system that can tolerate high latencies while preserving coherency; *find* moves part of the computation required to locate files of interest to the server. For each system described in this section, there are others not cited that either fall into the same kind of distributed file system or are close to the one described.

First, there are many network file system protocols successfully used in LANs, since long ago. We can mention NFS [3], and similar systems like the Common Internet File System, or CIFS [14], the Apple Filing Protocol, or AFP [15], 9P [16], and Styx [17]. Being designed for LANs, they do not handle high latencies well.

NFS departs from others in this group in that it now considers latency in its design. NFS has been the standard for more than two decades. It started as a stateless file system protocol and later evolved to include more RPCs and optimizations. Old versions required multiple round-trips and used UDP. The latest version, NFSv4, uses TCP and is no longer stateless. NFSv4 is used by some Cloud Storage providers, such as Amazon’s EFS (Elastic FS) [18]. It supports compound RPCs to deal with high latency. The client batches operations and the server batches responses, so the number of RPCs is greatly reduced.

However, a problem with the NFS approach is that an error in one of the operations makes the server reject the rest of the batch, and the client has to resend operations not yet performed. Unlike NFS and related systems, ZX includes *find* as a mean to batch the whole series of requests that would be necessary to locate files of interest. In most other cases, batching is used to coalesce reads and writes, which ZX does not require because it relies on streaming-RPCs. Thus, error handling is cleaner for ZX.

SSHFS is based on the SSH's Secure File Transport Protocol [19]. Most SSHFS implementations (i.e. OpenSSH) are based on SFTP version 3 [20]. SFTP is based on synchronous RPCs, following the UNIX API for file operations. As a result, SSHFS also requires multiple roundtrips in many cases. Some efforts are being done to improve its performance by using windowing [21]. Unlike SSHFS, ZX combines RPCs and streaming by using channels for requests and replies. Also, it has *find*.

VisageFS [22] takes a middle stand between loosely connected trees and full coherency, with three levels of consistency. In order to make metadata access faster, it provides modes of relaxed consistency as latency increases. In the worst case, the changes are made locally (in a hierarchy) and synchronized and reconciled later. Instead, ZX tries to preserve coherent access by relying on a different interface. Nevertheless, the ideas from VisageFS may be applied to ZX caches.

LBFS is a distributed file system designed for low bandwidth [23]. Its techniques are complementary to our work: LBFS considers a limited bandwidth environment, while ZX considers latency instead. LBFS requires, for example, at least two RPCs to read a file not in the cache, while ZX does not. That is reasonable for LBFS because it considers bandwidth as the problem and trades transferring data with extra RPCs.

HTTP, WebDAV [24], FTP, and similar protocols transfer whole files. They are similar to ZX in that they are able to stream data, but they differ in that ZX is able to operate on parts of files and also in that they lack *find*.

Since version 1.1, HTTP includes pipelining capabilities [25,26]. Pipelining requires both servers and clients to support such feature and permits multiple requests to be issued without waiting for a reply before issuing another request. This is similar to the multiplexing done by ZX and others in that a single TCP connection may be multiplexed among multiple ongoing requests (i.e., message tags are used to match replies to requests, or to identify channels). ZX differs in that it is designed as a file system and its requests are closer to the UNIX file interface than they are to HTTP. Also, it has *find*.

Another feature introduced since HTTP/1.1 is persistent connections. It permits clients to keep open connections to servers for further requests, instead of opening a different connection for each request. As a consequence, it is practical to use multiple connections in HTTP clients at the same time, for example, to retrieve multiple documents. Although there is nothing that forbids ZX from using multiple connections to the same server at the same time, the current implementation uses a single one. Using multiple connections can improve the effective bandwidth for data transfers. Regarding latency, the key point is that streamed data is sent as a series of messages, and that the connection is multiplexed among messages with a limited size. Therefore, further request and reply data may be sent in the middle of existing transfers. We plan to explore parallel connections and some bandwidth-related optimizations done by others (eg. SSHFS) as future work.

Op [27,28] is designed to transparently bridge Styx [17] over high latency links. It is a 9P [16] descendant. Op relies on two requests to put and get both data and metadata. The data is carefully cached in the client's adaptor in order to support synthetic files (a core abstraction in Inferno). Op (and also Styx and 9P) are closer to the UNIX interface than ZX is, and neither use streaming-RPCs nor include a *find* request.

Chirp [4] is a file system protocol designed to improve performance when accessing files in Grids. Like ZX, it combines RPCs with multiple responses to read and write files. To transfer small files, RPCs are used. To transfer large files, streaming is used instead. Like Thain et al. observed, the limitation is imposed by the UNIX file API, thus two streaming operations (*get* and *put*) were proposed as an addition. Native Chirp tools use their own library to take advantage of these new operations. Chirp also includes some optimizations, such as third party transfers, listing a directory (entries and their metadata) with a single RPC, and recursive deletion. UNIX applications use a FUSE server to access files. Although it is close to ZX, ZX differs in that it does not change the communication mechanism depending on the file size, thanks to channel-based streaming RPCs. Besides the *removeall* request providing recursive deletion, ZX includes *find*, which batches recursive inspection requests in most of the cases.

Clustered and parallel file systems for HPC, like GoogleFS [29] and Lustre [30] focus on parallel computing, which is a different issue not addressed by ZX at all. They are designed for availability, scalability, and bandwidth usage on clusters connected through low-latency networks. Instead, ZX focuses on high-latency links and, although the separation of finders and file trees enables parallel computing, it is not clear how ZX would perform in parallel computing environments. Zebra [31] is a system that uses a dedicated server for metadata and data striping. However, file striping does not improve the performance of interactive usage of the filesystem when the latency is very high. In ZX, the bottleneck considered is not system I/O, but network latency.

Lustre is designed to transmit huge amounts of data between data centers through WANs. Its filesystem protocol batches RPCs and uses windowing (called *RPCs in flight*) to support high latency. They consider an RTT of 4 ms as very high latency [32]. ZX differs in that it addresses higher latencies (above 50ms, for example), and also in that, using *find*, it avoids the need to batch RPCs for inspecting the file system. Also, ZX uses streaming-RPCs instead of batching them.

Well-known distributed file systems like, for example, AFS [33] and Coda [11] rely on a local cache for remote trees. Coda goes further and permits disconnected operation. When the server becomes reachable, the cache is synchronized. ZX does not require caches, although it cannot support disconnected operation on its own.

Orifs [2] provides peer-to-peer, transparent, replication of trees. Unlike ZX it relies on change propagation and conflict resolution. The archival system is built into Orifs. In ZX an external program is used.

Cloud storage systems, like Dropbox [34], distribute trees among devices. The Dropbox client executes a protocol to reach Dropbox servers for metadata and data, in order to keep the local replica up to date. The Dropbox REST API provides operations to upload/download complete files, manage the namespace (copy, move, and delete full files and directories in the namespace), and restore previous versions of files. Coherency is sacrificed in favor of availability, as in other Cloud systems. ZX is closer to distributed file systems described above than it is to Dropbox and related systems.

Systems like Git [35] and Mercurial [36] can be used to replicate trees at different machines, and to synchronize them manually. Tools like Rsync [37] synchronize disconnected trees. ZX takes the opposite design space and relies on a central, coherent, file server. It differs from them mainly because the problem addressed is different. They focus on user mobility rather than on enabling interactive usage of a remote file system. Rsync includes many optimizations for minimizing data transfers that may be applied to data transfers in ZX; although they are not yet implemented.

Much work has been conducted on asynchronous and streaming RPCs [38,39], for file systems and for other purposes. Asynchronous RPCs do not block the caller and, in general, permit the caller to retrieve the result later on. Streaming RPCs as used by ZX permit data to be streamed as part of the *call* for the RPC, and also as part of the reply. That is different from asynchronous RPCs used by others. The Chirp [4] system described above relies on a streaming RPC for its streaming mode of data transfers. ZX differs in that its RPCs are channel-based and a generic multiplexor is used to make all the requests and replies become a channel. They are to the Bell-labs style of concurrent programming (exchange data through channels to communicate) what the RPCs are to a procedure calls.

Scriptable RPCs (or SRPCs) [39] enable the clients to send RPC-scripts to the server to extend it with new operations. Therefore, ZX *find* might perhaps be implemented using SRPCs. However, doing so, requires RPC scripts to accept looping constructs and that is a safety issue (as authors of [39] admit). On the other hand, forbidding looping constructs increases security but prevents implementation of a *find* request. In ZX, *find* is a built-in operation and therefore it does not raise the security issues raised by scriptable systems. Also, SRPCs are still RPCs in that they build a result and deliver it back to the client, and it is not clear if they might support streaming as ZX does.

The file system from the Speculator project [40] relies on asynchronous RPCs, validated before returning control to the client, to provide a UNIX-like interface while supporting asynchronous execution. The main difference is that ZX operations are synchronous, albeit streaming, and thus the error handling mechanisms required by applications are simpler in ZX that in the Speculator (and similar systems). Also, ZX includes *find*, which is not supported by the Speculator file system.

An important difference is that errors may be forwarded through the reply channel for a ZX RPC in-line with other reply data. For example, *find* issues errors for individual files (eg., "permission denied") along with directory entries. The client code can cleanly process both errors and directory entries received. Source code for this was not shown in the examples for brevity, but there are many examples in the Clive source code.

5. Evaluation

This section includes some evaluation results for ZX for both using the FUSE adaptor and using the raw ZX interface. Because ZX adopts a different interface, it is hard to present quantitative results that could be fair for both ZX and other systems. Using different interfaces has profound implications for the performance of the task being measured.

We were unable to build and run a few of the systems described in the related work section. But this is understandable, because they are research systems published some time ago, and they are not expected to have support and/or remain operational for long. It is likely that in a few years ZX will have problems to be built and used, should it be replaced with another research system at our laboratory.

Considering this, and that systems like NFS and SSHFS are well-known, and provide a common ground for performance evaluation, we compare ZX to them. The measures shown here should be taken more as a qualitative indication of the relative performance than as an exact measurement of its value.

The next section presents the worst case for ZX (no extra latency) and presents the result of standard *FileBenchs* for ZX, SSHFS, and NFS. This illustrates the overhead for ZX when compared to others in a good scenario for NFS. Then we present some macrobenchmarks that illustrate how ZX may outperform such systems on the scenario addressed by this work.

We used the latest versions of SSHFS (v2.5) and NFS (v4) as distributed with Linux, and a FUSE v2.9.2 driver. That is to say that both systems include changes made to better handle latency issues. For example, NFS version 4 knows how to batch requests and exploits state kept in the server, unlike earlier versions of NFS. In particular, this version of NFS is capable of walking paths with multiple name elements (previous versions required at least one RPC per path element).

The numbers show that ZX performs well under high latencies, and that its performance is reasonable (albeit worse) for a local area network.

All the experiments were conducted on a machine running Linux 3.13 SMP X86-64, using 4 Xeon 2.1Ghz cores, 4 Gb of RAM and a 160GiB wd1602abks-18n8a0 SATA hard disk. Most of them were repeated on a Supermicro AS-1042G-TF Opteron 6128 with 32 cores (8 per socket), using a 1TiB 7200RPM ST31000524NS SATA hard disk. In all cases, the results were similar.

5.1. Microbenchmarks

Filebench is a well-known synthetic microbenchmark [41,42]. It is a file system performance evaluation framework widely used in the literature that may operate under different personalities. Here we used the *networkfs* personality, which is a set of micro workloads for measuring networked file systems. The benchmarks described here operate on a fractal file tree using a *fileprint* of 270MiB and a total of 1100 files. In short, workload *rmw1* reads and creates some files, removing one of them. Workload *launch1* reads

multiple files. The numbers reported for each workload by *FileBench* try to measure the performance of individual operations. Being a microbenchmark, the important point is that results for different file systems provide a measure of their relative performance for particular operations made by the benchmark. Tables 2 and 3 show the bandwidth and operations per second for *Zxfuse*, SSHFS, and NFS.

	SSHFS	**Zxfuse**	**NFS**
Bandwidth	3.2 Mb/s	1.2 Mb/s	3.2 Mb/s

Table 2: *Filebench bandwidth for SSHFS, Zxfuse, and NFS. Using the loopback, with no extra latency.*

	SSHFS	**Zxfuse**	**NFS**
rmw1.deletefile1	60 ops/s	22 ops/s	60 ops/s
rmw1.closefile2	60 ops/s	22 ops/s	60 ops/s
rmw1.writefile2	60 ops/s	22 ops/s	60 ops/s
rmw1.newfile2	60 ops/s	22 ops/s	60 ops/s
launch1.closefile5	10 ops/s	4 ops/s	10 ops/s
launch1.readfile5	10 ops/s	4 ops/s	10 ops/s
launch1.closefile3	10 ops/s	4 ops/s	10 ops/s

Table 3: *Filebench operation results for SSHFS, Zxfuse, and NFS. Using the loopback, with no extra latency.*

Measurements include the impact of the UNIX kernel and its internal caching, but were made with the kernel in the same conditions for each measurement. Also, *Zxfuse* asks the kernel not to cache its files (as far as it can ask). Also, the UNIX kernel is asked to drop caches before each experiment. This is done to minimize the effect of the kernel caching in the measurements.

This is the *worst case* scenario for ZX: no extra latency and using the UNIX interface for file operations through the FUSE driver. Also, *Zxfuse* has not been optimized (while NFS and SSHFS have been) and makes two to three times more system calls than strictly required due to the FUSE adaptor.

Taking this into account, it is reasonable that NFS and SSHFS outperform ZX for this particular experiment. However, the results also show that ZX is on the same order and that it can be used in practice even for a well-connected local area network. Although such a version does not exist (yet), we expect an optimized version of *Zxfuse* to perform closer to SSHFS. But, in a very-low latency scenario, ZX is not the best choice in most cases.

The macrobenchmarks shown next show better scenarios for ZX, and consider the effect of departing from the UNIX file interface.

5.2. Macrobenchmarks

Figure 5 shows the result for finding an existing file name on the file system. This is a very good scenario for ZX because it matches its *find* operation, which is not available in NFS or SSHFS.

For NFS, SSHFS, and *Zxfuse* we use the UNIX *find* command. In this case, to be fair, the *Zxfuse* driver uses a write-through cache. Of course, the UNIX kernel cache is the same in all the cases. NFS runs unencrypted and is implemented in the kernel, which is more realistic, but this is also unfair for others.

For ZX (with the native ZX interface), we use the Clive *lf* command, which is similar to the UNIX command used, but speaks ZX. In this case, *lf* issues a single *Tfind* request (with no further data in the request channel). Upon reception of such request, the server streams entries one after another through the reply

channel. On the other hand, *Zxfuse* walks the tree as the UNIX kernel walks it, but issues a single *Tget* for each directory visited which permits the server to stream its contents back to the ZX FUSE driver. NFS batches directory entries at the same level in some cases (depending on the actual calls made within the kernel).

The results show a speedup resulting for ZX and *Zxfuse*. When latency goes from 0ms to 50ms: execution times for ZX go from 4.2s to 5.2s; times for *Zxfuse* go from 43.168s to 619.5s; times for NFS go from 1.2s to 1231.3s; and times for SSHFS go from 3.9s to 706.2s.

Once more, with no latency, NFS and SSHFS outperform ZX and *Zxfuse*. As latency increases, things change and both ZX and *Zxfuse* outperform others. Using the native ZX interface is of course the best case, because ZX was designed just for this case. For example, with latencies above 20ms, ZX presents speedups of more than two orders of magnitude. This is the effect of (1) downloading the find task to the server and (2) streaming replies to the client. When using ZX through FUSE, the streaming underlying ZX compensates for the overhead of the FUSE interface as the latency increases.

Although this is the best case for ZX, in practice, the early phase of execution for many commands is similar to *find*: most commands have to locate some files to do their work. Therefore, the benefits might apply to other commands as well, at least in part.

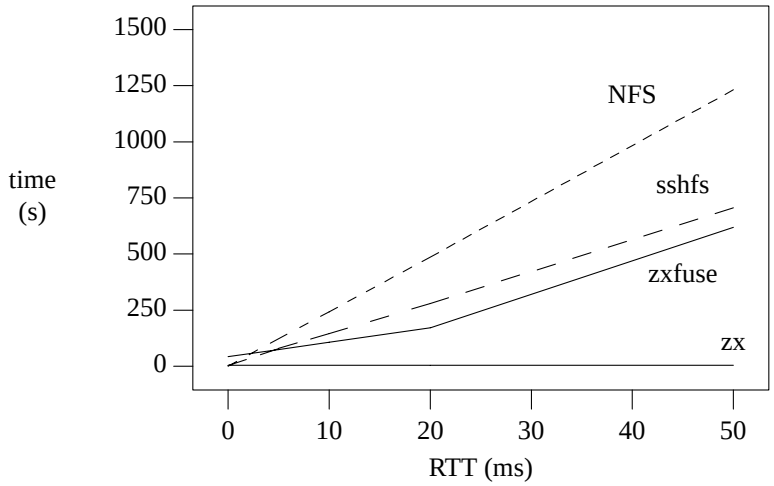

Figure 5: *Find files with cold cache as latency increases, for ZX, ZXfuse, SSHFS, and NFS.*

Figure 6 tries to isolate the effect of kernel caching in the same setup. It presents the results for issuing a second find to search for files that do not exist (after the find measured in the previous experiment). That is, the cache is warm in this case. The find request searches missing files to measure the effect of the protocol and the file interface, instead of measuring just the cache. Also, only NFS and ZX are included this time, to remove the effect from user caches from the FUSE drivers in *Zxfuse* and SSHFS.

In general, results for ZX and NFS are similar to those shown before. NFS performs better because many directory entries are found in the cache, but RPCs still add their times quickly. The warm cache is better than the cold one, as expected.

Figure 7 is the result for another experiment: removing all files found under a given directory. In this case multiple RPCs are required for both NFS and ZX to actually remove the files found.

When used directly, ZX is orders of magnitude faster than *Zxfuse*, SSHFS, and NFS. This is the effect of

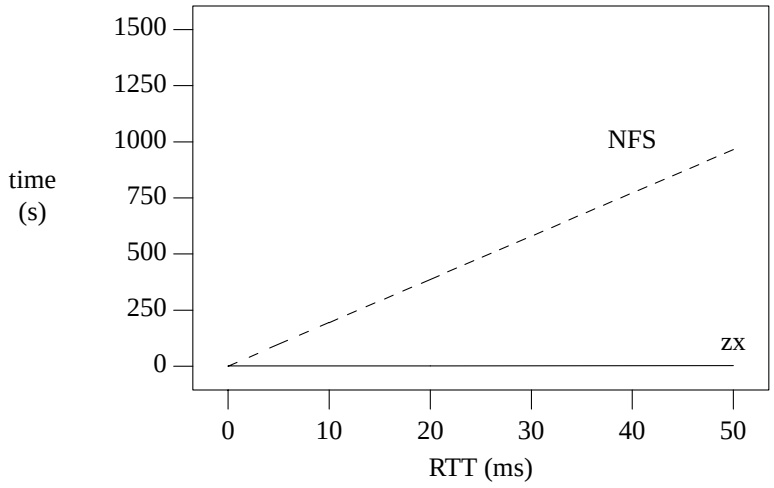

Figure 6: *Find missing files with warm cache and no FUSE adaptor as latency increases, for ZX and NFS.*

(1), having *find* as an operation to stream directory entries of interest back to the client and, (2) being able to issue file operations for them while the stream is being processed.

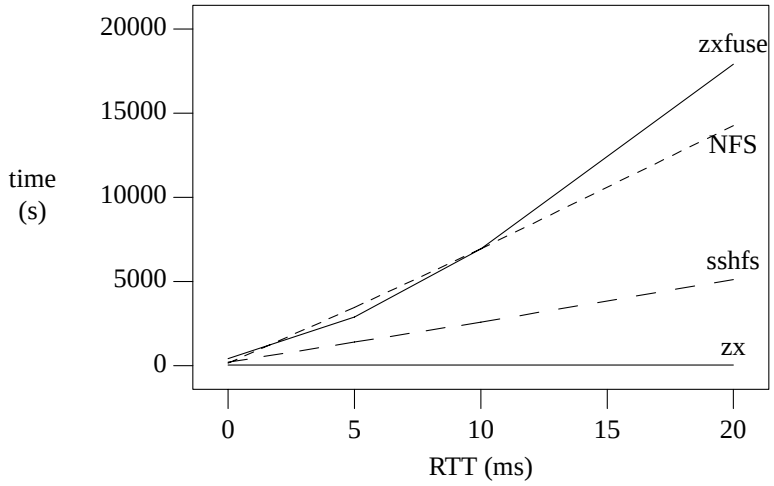

Figure 7: *Removing all regular files for ZX, ZX through FUSE, NFS, and SSHFS as latency increases.*

When used through the UNIX API with FUSE, ZX is still competitive until its overhead for the multiple RPCs made to remove the files goes over the speedup resulting from the streaming of entries for files to be removed. For 20ms of RTT, it is three times slower than SSHFS. But note that even up to 10ms the effect of streaming in the find phase of the execution compensates for the overhead of the FUSE adaptor.

When using the ZX interface it is easy to issue remove requests as we receive entries for files to be removed, and the execution time for 20ms remains excellent. In this particular case, the client issues a single *Tfind* request and the server replies with a stream of matching directories. As soon as the client receives streamed data, it issues *Tremove* requests which may be processed concurrently (both with other remove requests and with the ongoing *find*). Each *Tremove* request is replied with a completion status which is kept in the reply channel buffer at the client until the client code consumes it.

On the other hand, all other file systems must walk the tree and issue separate remove requests. SSHFS heavily relies in its cache for walking, which makes it run fast for this experiment (but not faster than speaking ZX). NFS cannot walk multiple path elements at a time in this case, but it may batch requests and performs better than *Zxfuse*. *Zxfuse* has to honor the calls made from UNIX through FUSE and must wait for stat and remove requests to complete before replying to the kernel, becoming slower when latency increases in this case.

Figure 8 presents another experiment where files are actually read. It computes differences for two file trees using *diff* (on the source of Linux's `src/crypto` as the first file tree, and the same tree with an "`a`" appended to each file as the second tree).

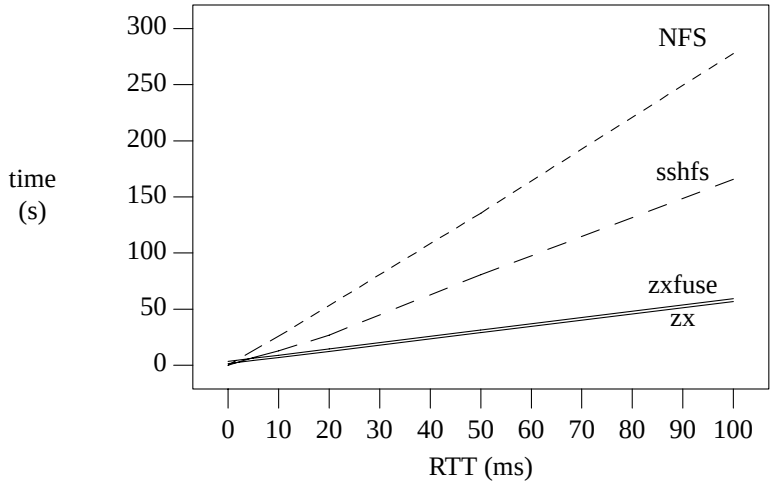

Figure 8: *Computing file differences in a small file tree for ZX, ZX through FUSE, NFS, and SSHFS as latency increases.*

Again, it seems that the interface changes and the protocol proposed by ZX lead to a significant speedup with respect to SSHFS and NFS. Because the number of directories is small, batching for SSHFS and NFS implies only a few operations, but even so, streaming seems to performs better. Also, the larger the number of directories, the better for ZX.

In this experiment, the FUSE cache waits until files have been received, and the net effect is that for ZX there is not much difference between its native interface and using the FUSE adaptor. But note that in the case of the FUSE adaptor for ZX, it is still the ZX interface the one responsible for the speedup of *Zxfuse* with respect to SSHFS and NFS. It speaks ZX to the remote file server, which leverages streaming for fetching files. Both the native ZX and *Zxfuse* issue a *Tget* per file or directory to retrieve its contents, which are streamed back to the client. Others read file and directory contents with some degree of read-ahead (as dictated by the UNIX kernel in the case of NFS and by the user-level cache in the case of SSHFS). Each read implies one round-trip.

The next experiment performs a recursive *grep* for file contents in a Linux kernel source tree. Figure 9 presents the results when using a cold cache and figure 10 shows the results for a warm cache. In both cases, ZX is several orders of magnitude faster than their counterparts, with just 10ms of latency. For zero latency it is slower, as it could be expected. Messages issued are similar to the ones in the previous experiment.

Another experiment compiles the SSHFS source tree using SSHFS, NFS, and *Zxfuse*. This usage combines both file reads and file creation and rewrites. The ZX FUSE driver issues a *Tget* per file or directory visited

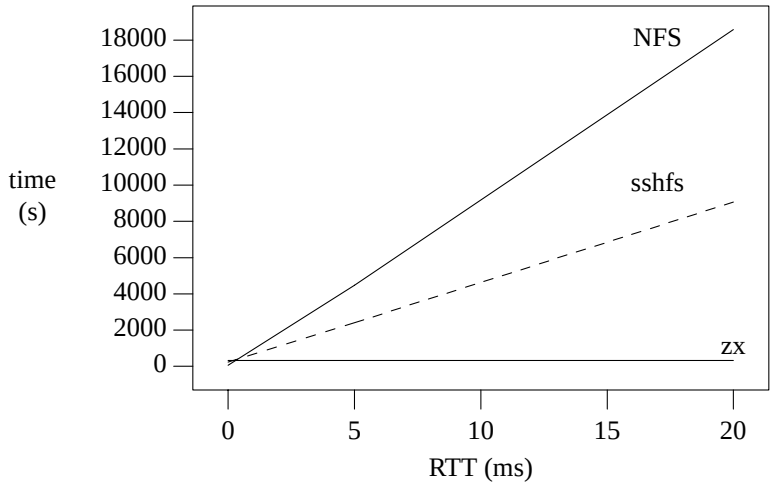

Figure 9: *Grep for files for ZX, NFS, and SSHFS as latency increases with cold cache.*

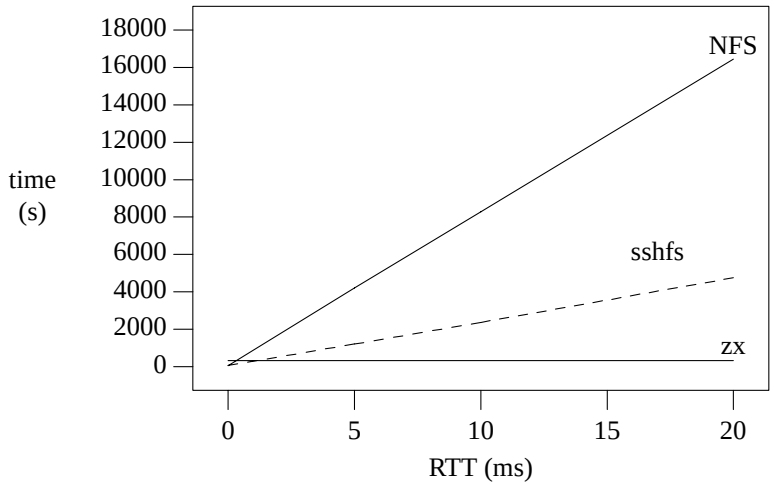

Figure 10: *Grep for files for ZX, NFS, and SSHFS as latency increases with warm cache.*

(fetching its contents with a reply stream), and a *Tput* per output file written sequentially (most object files). For output binaries (which result from the linking phase), random access writes are often the case and multiple *Tput* messages are issued. Individual write requests made by UNIX are written through the request channels in all cases.

The experiment results can be seen in figure 11. NFS is faster than ZX with the FUSE driver for latencies under 10ms, as it could be expected. The reason is that NFS is an in-kernel optimized implementation matching well the kernel file system interface, and that the version used considers latency issues to some extent. Unlike NFS, ZX must go through the FUSE driver for this experiment. However, ZX outperforms NFS for higher latencies. It is also interesting to note that, unlike in previous experiments that are intensive in file reads, NFS outperforms SSHFS as well. One reason for this is that for writes and file creation NFS leverages the kernel cache with delayed writes but SSHFS and ZX must go through the FUSE driver (and its internal cache) instead. In general, we can say that for this kind of workload ZX is competitive and performs well for large latencies.

To evaluate the effect of a ZX file system stack (see figure 4), a few more experiments have been made. These experiments are exactly the same ones shown in figure 11 (compiling a file tree), figure 9 (running *grep*), and figure 8 (computing differences in a file tree) and the setup used for them is exactly as in the

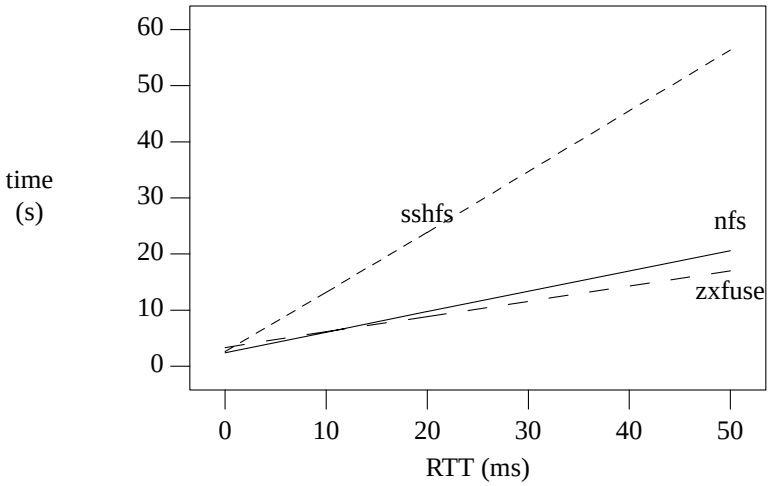

Figure 11: *Compile the SSHFS source tree for NFS, ZX with FUSE, and SSHFS as latency increases.*

previous ones. But this time the experiments measure the times for different ZX file system stacks. In particular, we measure stacks with all combinations resulting from including or not a client-side cache and including or not a server-side ZX cache. When both caches are included, the resulting ZX stack is the one depicted in figure 4.

Figure 12 presents the results for compiling the SSHFS source tree, figure 13 corresponds to the *grep* experiment with cold caches, and figure 14 shows results for computing file differences.

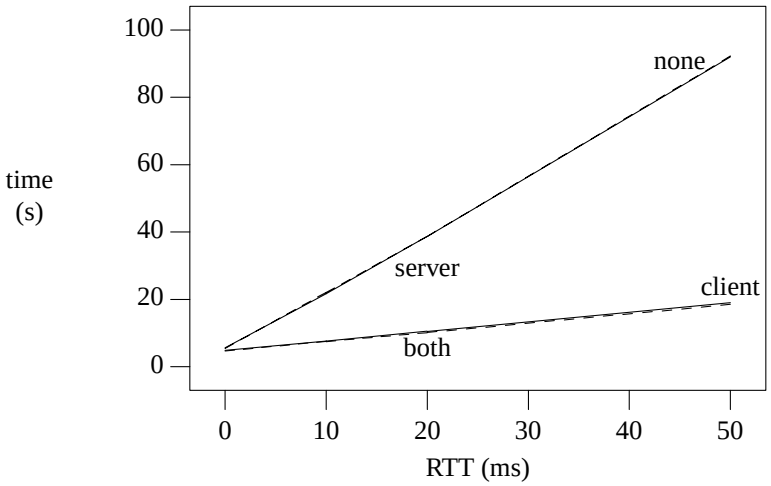

Figure 12: *Compile SSHFS for different ZX file system stack configurations as latency increases. Plots correspond to no caches (none), client cache only (client), server cache only (server), and both client and server caches (both).*

As it can be seen, using ZX cache at the client has a significant impact in performance. This is as it could be expected. Caching files in the client's memory implies that the latency required for reaching the files is insignificant once the files are cached. Also, the cache operates as a ZX client when fetching files from the ZX file system server. This means that the cache may stream entire directories from the server to fetch file metadata, and it may stream full files from the server as well. Therefore, when checking out if a file is up to date, near-by files are checked out as well. Moreover, the cache streams data to the server using *put*, which is efficient and avoids the need to block the cached files during multiple RPC round-trips when writing files through.

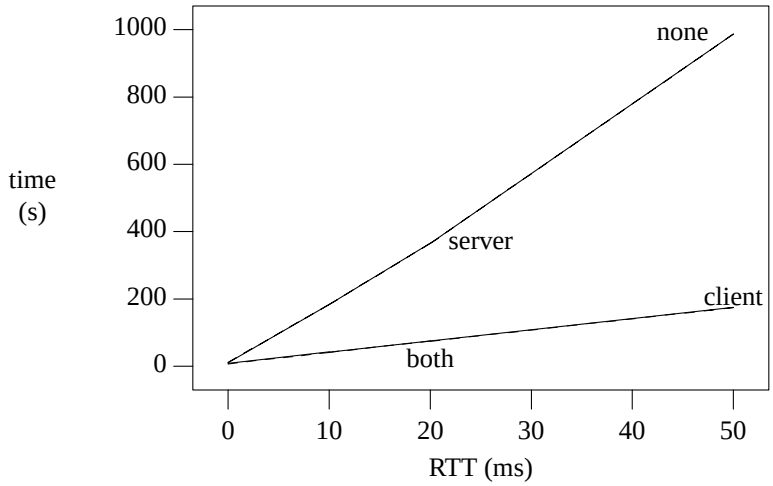

Figure 13: *Results for grep with cold cache for different ZX file system stack configurations as latency increases. Plots correspond to no caches (none), client cache only (client), server cache only (server), and both client and server caches (both).*

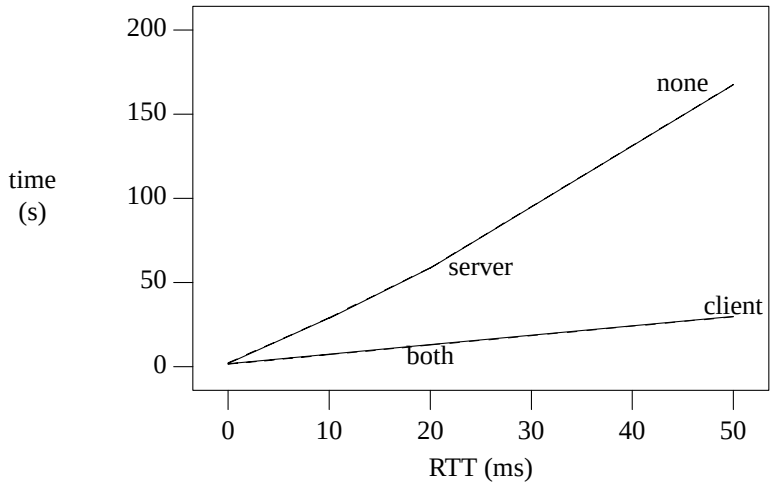

Figure 14: *Results for computing file differences for different ZX file system stack configurations as latency increases. Plots correspond to no caches (none), client cache only (client), server cache only (server), and both client and server caches (both).*

Using the ZX cache at the server makes the experiments run a little bit faster, but times are still close to those without a server cache. The reason is that the server's kernel caches disk blocks on its own and that disk latencies are insignificant compared to network latencies (as measured). In most cases, popular files and directory entries are already in the server's memory.

We may conclude from the experiments shown that the approach of using a central coherent server can be practical in the presence of some (or high) latency.

For a local-area network with low latency, NFS or similar systems might be a better choice if we consider just the execution times (because ZX provides coherent access to a central file tree while NFS and others also rely on caching).

Access through FUSE is slower, but we have to pay this price to leverage legacy UNIX tools that do not know how to speak ZX. Also, it may still be faster when finding files.

As latency increases, ZX can result in significant speedups. The combination of a *find* operation and operations that may stream data result in better performance.

6. Conclusions and future work

We have shown a new file interface and protocol for file access through high-latency networks with more than 50ms of round-trip times, and that it can be used for workloads typical in interactive system usage. Evaluation results indicate that using a central coherent server can be still practical in the presence of latency.

Using ZX fits well with CSP-like languages, because its interface is closer to such model than the one provided by UNIX and related systems.

We have also shown that, at a price, FUSE can be used to permit legacy applications to work through ZX.

We have been using ZX for months now, both on the LAN at work, and also when using it from home or away. Although figures shown indicate that NFS is a better choice for a LAN (regarding execution times), our experience says that ZX performs fine for actual usage in such a case.

This paper was written using ZX both to access the files for its source files and to run the programs used to format it and generate PDF output files. Latency (according to *ping*) was about 70ms most of the times, yet using remote files was as convenient for us as it was using a local disk. When used from a metropolitan or wide area network, ZX permitted us to continue using a central file server, instead of relying on synchronized (not coherent) local storage.

As future work we plan to conduct profiling and fine tuning, and to borrow for ZX optimizations introduced by others to reduce required data transfers. We plan also to evaluate how the implementation scales with respect to the number of clients using a server.

All the software and documentation, including the modified Go compiler, Clive source code, its user's manual, and the implementation for ZX, can be found at [http://lsub.org].

References

1. The Clive Operating System. Francisco J. Ballesteros. GSyC/LSUB TR 14-4. http://lsub.org/export/clivesys.pdf.

2. Replication, History, and Grafting in the Ori File System. Mashtizadeh, Ali Jose,Bittau, Andrea,Huang, Yifeng Frank,Mazi'eres, David. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. Pgs.151--166. http://doi.acm.org/10.1145/2517349.2522721. ACM. Farminton, Pennsylvania. 2013.

3. Network File System (NFS) version 4 Protocol. S. Shepler,B. Callaghan,D. Robinson,R. Thurlow,C. Beame,M. Eisler,D. Noveck. Internet RFC3430. 2003.

4. Efficient access to many small files in a filesystem for grid computing. Douglas Thain,Christopher Moretti. Grid Computing, IEEE/ACM International Workshop on. Vol. 0. Pgs.243-250. 2007.

5. Rule-based curation and preservation of data: A data grid approach using iRODS. Mark Hedges,Tobias Blanke,Adil Hasan. Future Generation Computer Systems. Vol. 25. Nb. 4. Pgs.446-252. Elsevier. April 2009.

6. The Open Group Base Specifications Issue 7. The Open Group. pubs.opengroup.org. IEEE 1003.1. 2001.

7. The Go Programming Language. The Go Authors. http://golang.org.

8. Communicating sequential processes. Hoare, C. A. R. Commun. ACM. Vol. 21. Nb. 8. Pgs.666--677. New York, NY, USA. Aug. 1978.

9. Lsub Go. Francisco J. Ballesteros. Lsub TR 15-3. http://lsub.org/export/golsub.pdf. 2015.

10. 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. Vol. 23. Nb. 7. Pgs.260-267. ACM. 1988.

11. Coda: A Highly Available File System for a Distributed Workstation Environment. M. Satyanarayanan,James J. Kistler,Puneet Kumar,Maria E. Okasaki,Ellen H. Siegel,David C. Steere. IEEE Transactions on Computers. Vol. 39. Nb. 4. Pgs.447-459.

12. Plan 9 from Bell Labs. R. Pike,D. Presotto,K. Thompson,H. Trickey. EUUG Newsletter. Vol. 10. Nb. 3. Pgs.2-11. Autumn 1990.

13. SSH Filesystem. http://fuse.sourceforge.net/sshfs.html.
14. CIFS; A common Internet System. Paul Leach,Dan Perry. Microsoft Interactive Developer. Nov. 1996.
15. Inside AppleTalk. Sidhu, Gursharan S,Andrews, Richard F,Oppenheimer, Alan B. Addison-Wesley Reading. - 1990.
16. The Organization of Networks in Plan 9. Dave Presotto,Phil Winterbottom. Plan 9 User's Manual. Vol. 2.
17. The Styx Architecture for Distributed Systems. Rob Pike,Dennis M. Ritchie. Bell Labs Technical Journal. Vol. 5. Nb. 2. April-June 1999.
18. Amazon Elastic File System. http://aws.amazon.com/efs/.
19. SFTP specifications. https://wiki.filezilla-project.org/SFTP_specifications.
20. SSH File Transfer Protocol. T. Ylonen,S. Lehtinen. RFC-Draft. https://filezilla-project.org/specs/draft-ietf-secsh-filexfer-02.txt. 2002.
21. Making SFTP transfers fast. http://daniel.haxx.se/blog/2010/12/08/making-sftp-transfers-fast/.
22. VisageFS dynamic storage features for wide-area workflows. Thiebolt, F., Ortiz, A., Mzoughi, A. Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems. 2007.
23. A low bandwidth network file system. Athicha Muthitacharoen,Benjie Chen,David Mazieres. ACM SOSP. Pgs.174-187. 2001.
24. WebDAV: a network protocol for remote collaborative authoring on the Web. E. James Whitehead,Yaron Y. Goland. Proc. of the 6th conference on Computer Supported Cooperative Work ECSCW99. 1999.
25. Hypertext transfer protocol - HTTP/1.1. Fielding, R.,Gettys, J.,Mogul, J.,Frystyk, H.,Masinter, L.,Leach, P.,Berners-Lee, T. RFC Nb. 2616. 1999.
26. Network performance effects of HTTP/1.1, CSS1, and PNG. Nielsen, H. F.,Gettys, J.,Baird-Smith, A.,Prud'hommeaux, E.,Lie, H. W.,Lilley, C. ACM SIGCOMM Computer Communication Review. Vol. 27. Nb. 4. Pgs.155-166. ACM. October, 1997.
27. Octopus: An Upperware Based System for Building Personal Pervasive Environments. Ballesteros, Francisco J.,Soriano, Enrique,Guardiola, Gorka. J. Syst. Softw. Vol. 85. Pgs.1637--1649. http://dx.doi.org/10.1016/j.jss.2012.02.011. Elsevier Science Inc. New York, NY, USA. 2012.
28. Op: Styx batching for High Latency Links. Francisco J. Ballesteros,Gorka Guardiola,Enrique Soriano,Spyros Lalis. IWP9. 2007.
29. The Google File System. Ghemawat, Sanjay,Gobioff, Howard,Leung, Shun-Tak. SIGOPS Oper. Syst. Rev. Vol. 37. Pgs.29--43. http://doi.acm.org/10.1145/1165389.945450. ACM. New York, NY, USA. 2003.
30. Lustre Filesystem. http://lustre.org.
31. The Zebra Striped Network File System. Hartman, John H.,Ousterhout, John K. ACM Trans. Comput. Syst. Vol. 13. Pgs.274--310. http://doi.acm.org/10.1145/210126.210131. ACM. New York, NY, USA. 1995.
32. HPC File Systems in Wide Area Networks: Understanding the Performance of Lustre over WAN. Aguilera, Alvaro,Kluge, Michael,William, Thomas,Nagel, WolfgangE. Euro-Par 2012 Parallel Processing. Vol. 7484. Pgs.65-76. http://dx.doi.org/10.1007/978-3-642-32820-6_9. Springer Berlin Heidelberg. 2012.
33. Andrew: A Distributed Personal Computing Environment. Morris, James H.,Satyanarayanan, Mahadev,Conner, Michael H.,Howard, John H.,Rosenthal, David S.,Smith, F. Donelson. Commun. ACM. Vol. 29. Pgs.184--201. http://doi.acm.org/10.1145/5666.5671. ACM. New York, NY, USA. 1986.
34. Dropbox. https://www.dropbox.com.
35. Git, fast version control. http://git-scm.com.
36. Mercurial. http://mercurial.selenic.com.
37. The rsync Algorithm. A. Trigdell,P. Mackerras. At http://rsync.samba.org. Australian National University.
38. A Survey of Asynchronous Remote Procedure Calls. Ananda, A. L.,Tay, B. H.,Koh, E. K. SIGOPS Oper. Syst. Rev. Vol. 26. Nb. 2. April, 1992.
39. Evolving RPC for active storage. Muthian Sivathanu,Andrea C. Arpaci-Dusseau,Remzi H. Arpaci-Dusseau. ACM SIGPLAN Notices. Vol. 27. Nb. 10. 2002.
40. Speculative Execution in a Distributed File System. Edmund B. Nightingale,Peter M. Chen,Jason Flinn. ACM Transactions on Computer Systems. Vol. 24. Nb. 4. November 2006.
41. The new and improved FileBench. Wilson, A. Proceedings of 6th USENIX Conference on File and Storage Technologies. USENIX. February, 2008.
42. Filebench tutorial. McDougall, R.,Mauro, J. https://koala.cs.pub.ro/redmine/attachments/download/603/filebench.pdf. Sun Microsystems. 2004.