ZX: A network file system for high-latency networks

Francisco J. Ballesteros
Gorka Guardiola
Enrique Soriano

Laboratorio de Sistemas (LSUB),
Departamento de Sistemas Telemáticos y
Computación (GSyC), Universidad Rey
Juan Carlos, Camino del Molino,
Fuenlabrada, Madrid, Spain

Abstract

Using a central file server is good for interactive access to files, because of the coherency implied by a centralized design. In fact, within local area networks, this is a common case. However, distributed environments in use today may exhibit round-trip times on the order of 50 or 100 ms. This is a problem for interactive file access to a central file server because of the resulting access times. Although aggressive caching and loosely synchronized replicas may be used for distributed file access, there are cases where the better coherency provided by a central server is still desirable. In this paper, we present ZX, a distributed file system and protocol designed with latency in mind. It can use caching, but it does not require caching or batching to address latency issues. ZX relies on a novel channel-based file system interface. It includes find requests and leverages streaming requests to work well under high-latency conditions. Unlike other protocols designed for distributed access to a central server, ZX tolerates round-trip times on the order of 50 or 100 ms to access a central file server for interactive usage such as compiling shared sources, running binaries, editing documents, and other similar workloads. It can be used on UNIX using a FUSE adaptor while permitting native ZX speakers to run faster.

KEYWORDS: cloud computing, network file system, operating system

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

1. INTRODUCTION

The Network File System [1] (NFS) has been the de facto standard in providing distributed access to a file tree. Although it works well with latencies as found in local area networks (LANs), that is not the case when wide area networks (WANs) are involved. 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 round-trip time (RTT), the common one being an overseas connection. For instance, the RTT from our University in 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 one-third lower than in the vacuum. Furthermore, routers (perhaps congested), switches, and other intermediate machines have to be taken into account. This latency can be worse over a wireless or a fourth-generation link when there is enough noise or when the signal strength is small. Many distributed systems, for example, that in the work of Mashtizadeh et al [2] ignored this issue because they were designed for local or metropolitan area networks. In fact, the NFS1 was designed initially as a file protocol for LANs and later evolved to consider network latency and perform better with worse latencies.

As latency increases, remote call round trips quickly add up and degrade the service. With high enough latencies (eg, RTTs of 50 ms or more), the file service may be slow enough to become inconvenient for actual file access. For example, using the NFS through a WAN might work, but it would be slow enough to cause timeouts in the kernel. Therefore, most users would consider the service as unavailable because of its (poor) performance.

As a result, other approaches have become popular for sharing data. For example, Erlang [3], Go [4] and other related systems adopt channel-based communication and a communicating sequential processes [5] (CSP)-like style to leverage streaming for distributed computing applications. However, these and other related approaches favor writing ad hoc software for the problem at hand instead of using existing software that knows how to use files for sharing resources.

There are other strategies for sharing files, such as those using loosely coherent replicas that rely on a synchronization protocol (eg, Dropbox [6]), that tolerate high latencies. The problem is that, despite their efforts, using a central file server may be better in terms of coherency for file access, and coherent file access is still a desirable property for some applications such as interactive access to file data.

In this paper, we describe ZX, which is a file system for a new operating system, namely Clive [7], built to permit the construction of efficient distributed services. ZX is an effort to make a central file server work well under bad latencies to provide better coherency for access to remote files. It supports file access through LAN and WAN links, which exhibit a wide range of latencies. 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 100 ms 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, and other similar workloads. The results shown later, as well as our own experience using this system through poor network links, support this claim.

We believe that there are 2 reasons that make it a problem to access a central file system through high-latency links:

- Using the UNIX file system interface and related system calls
- Using remote procedure calls (RPCs) to provide the service

Many file systems and protocols are 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 round trips from the application to the file server. As others have noted [8] a better interface is required for WANs. ZX departs from the venerable UNIX interface for file access. It uses channels and relies on FUSE adaptors to let UNIX applications work.

In addition, many file systems rely on RPCs to provide their services. However, latencies add up quickly when using RPCs for remote access because the client code must usually wait for 1 RPC to complete before issuing another. The result is that RTTs add up quickly, and when latencies go above 10 ms and become closer to 100 ms, using the resulting protocol becomes unpractical.

To overcome this problem, ZX applies 3 ideas:

1. 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.

2. Requests have channel-based interfaces suitable for streaming data both to the server and from the file server.

3. 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 guidelines permit programming and using applications that perform file access in a convenient way and still perform well under bad latency with good coherency. The resulting file interfaces are adequate for use in CSP-like languages such as Go, although they may be adapted to a UNIX-like interface. The main drawback of ZX is that, by design, it relies on a central file server, and network outages forbid file access. In those cases where disconnected operation is preferred to coherent file access, using a cache or another protocol might be a better choice.

The contributions of this paper are as follows:

• A new file system interface and protocol that performs well for distributed access to a central file tree through high-latency links
• A description of its implementation and examples of use in CSP-like languages
• A description of adaptors and tools that enable existing software to use it on UNIX
• A quantitative evaluation for the proposed system

The main contributions are how channels and a new find primitive are used to provide a file system interface and an implementation that tolerates high-latency links and is practical for actual usage. 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 also describe 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 AND PROTOCOL

The procedure calls for ZX requests accept input channels when they require data to be sent to the server, and always return reply channels. A reply channel is used after the call to retrieve both the status of the request and any data or error indication from the server. This interface is as easy to use as the UNIX-style interface for file access, but the underlying streaming helps with latency by improving RTTs as seen by the user. Furthermore, the interface provided is closer to languages that follow a CSP style (eg, Go and Erlang) than the UNIX interface is. This is important because such languages are often used for distributed and cloud computing applications.

A central request in ZX, ie, find, permits the client to issue a predicate to find files. As a result, many round trips to the server are avoided to locate files of interest. Many applications access files and directories to determine if they meet a desired property before using them. In this case, a single find request can stream the desired directory entries back to the client in a single request.

With bad latencies, this feature makes the difference between being able to use the file system or not (ie, or going for an alternative that sacrifices coherency). The drawback of relying on find is that an interruption of the 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.

In ZX, a directory entry for a file (or its metadata record) is a set of name/value attribute pairs. Their semantics are left out of the protocol in many cases. Attributes such as 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 such as iRODS [9] and POSIX extended attributes [10] permit arbitrary name/value attributes. Unlike them, ZX metadata follows the convention that attributes with names starting with an uppercase are not to persist in the file system storage. Therefore, programs may decorate directory entries by adding their own “temporary” attributes. For example, caches may add checksums and cache time stamps, and errors for file access may be noted in directory entries sent to other programs.

2.1 Clive overview

Clive [7] is a new operating system built for enabling the construction of simple and efficient cloud services. Although this paper presents ZX and not Clive, ZX has been built as part of Clive, and this section provides an introduction to Clive as an aid for understanding ZX.

Clive is a system built out of services interconnected through channels, which may cross the network. Processes in Clive run on other host operating systems or in an experimental native kernel (which is still a work in progress).

The central part of the system is a file system exported through a novel file system protocol, ie, ZX, which is presented in this paper. Such protocol is used to export not just regular files but also services exported as files (as in Plan 9).

In Clive, commands are implemented in a CSP-like style, and use named channels for I/O. Channels carry streams of typed data, including directory entries, raw bytes, and other data depending on the command. Such streams may carry full file trees, processed by command pipelines, without

requiring each command to perform RPCs to file servers. The results are improved performance, because of network latency, and greater flexibility because commands may define their own messages and talk through a pipeline without disrupting the data streams.

Clive processes include a per-process name space that maps path prefixes to finders, a path for the current working directory, a set of open I/O channels, and a set of environment variables. A Clive process may share these with other processes or keep a private copy for itself. Although there are differences, this is similar to the process architecture in systems such as Inferno [11].

The system is written in the Go programming language, using a modified Go compiler that implements the required support for Clive.

2.2 Channels

Both ZX and Clive are written using a modified Go [4] compiler. The Go language follows the style of CSP [5]. To be precise, Go follows a Bell Labs concurrent style, where processes exchange data through channels. The interface provided by ZX follows this style, as described next. The modified Go compiler [12] used by Clive permits senders and receivers to close a channel with an error indication that may be retrieved using a new cerror primitive.

Following is the idiom in Clive for receiving data from a channel (inc) and sending it through another channel (outc).

```
var inc, outc: chan of bytes
...
while receive data from inc {
   send process(data) to outc
   if send failed {
      close inc with cerror(outc)
      break
   }
}
close outc with cerror (inc)
```

In the code examples, the pseudocode shown is almost the actual code used in practice (which has the same number of lines). However, we changed the syntax and keywords to make the code more readable because the Go syntax may be cryptic at times.

This scheme of operation models 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. To abort a request, its channel is closed, and any sender gets an error when trying to send anything as part of the request. The output channel is also closed upon errors. A sender might have sent quite a bit of data before it notices that a request has been aborted, but the wasted bandwidth is part of the price for avoiding extra latency in the common case when the operation is performed with- out errors.

2.3 Finders and file trees

Unlike most other file systems, ZX splits its interface into 2 different entities.

• A finder interface, used to find directory entries
• A file tree interface, used to operate on files

A ZX Finder is anyone implementing the following interface.

Find(path, pred, spref, dpref: string, depth0: int): chan of 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.

It is important to note how this request makes ZX depart from the NFS and other systems discussed later in this paper. Because of the predicate given, the user may ask for a stream of matching directory entries. This could be used to stream entries for a whole (sub)tree or to stream those that have a particular property involving its name, path, size, permissions, or any other expression involving file metadata.

Considering that find takes the information needed to know the file path as seen by the user (by using the spref, dpref, and depth0 arguments), it is feasible to issue a find request for a mounted file tree that prunes the find at mount points within that tree at the client machine. Now, unlike in the NFS, it is feasible to concurrently find files on mounted file trees without having to go through one path element at a time (or a few at a time) because of mount points.

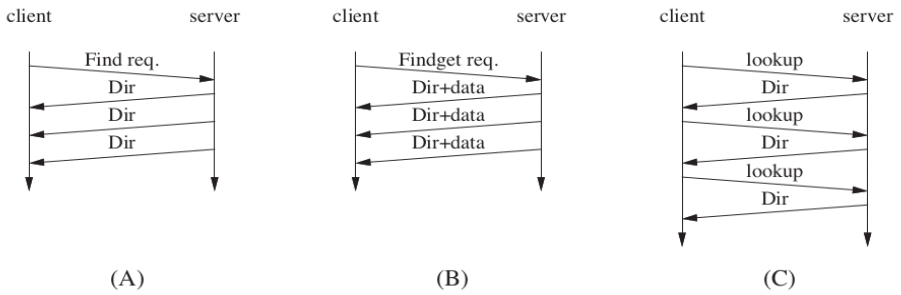

FIGURE 1. Sending multiple messages per request and reply leverages streaming and minimizes the effect of network latency in execution times. A, Find call; B, Findget call; C, Interaction using UNIX-style remote procedure calls (time flows down)

A related operation, findget, operates in the same way but returns both metadata and data for matching files through the reply stream, as shown in Figure 1.

FindGet (path, pred, spref, dpref: string, depth0: int): chan of any

This request exists because ZX is very aggressive to avoid round trips to the server. It avoids the need to call get for all files located by a previous find request. The result retrieved from the output channel is a series of directory entries for each matching file, with file data following each directory entry.

The interface for a ZX file Tree provides the following requests (or a subset).

Stat(path: string): chan of Dir
Get (path: string, off, count: int): chan of bytes
Put (path: string, d: Dir, off: int, dc: chan of bytes): chan of L
Move(from, to: string): chan of error
Remove (path: string): chan of error
RemoveAll (path: string): chan of error
Wstat (path: string, d: Dir): chan of Dir

Return channels are used very much like promises or futures [13] 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 such as 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 round trips 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 SHA-1 for file contents if they are provided by the file system implementation, or editors might use them to detect if a file was externally modified before saving its contents. 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 1 GB 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.

2.4 Implications for latency

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" (ie, the file type must be a directory) and "name *.go" (ie, the file name matches the given globbing expression, that is, the file name suffix is ".go").

The implementation 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, as follows.

```
dirc:= Find("/sys/sre", "type=d", "/", "/", 0)
while receive dir from direc {
    print (dir["path”])
}
if cerror (dirc) nil {
    warn (cerror (dirc) )
}
```

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 such as the NFS may look similar, but they are rather different from the ZX interaction. For example, for a single directory, the NFS may also stream directory entries back to the client. However, note that find may traverse a full file hierarchy and may filter entries of interest using its predicate.

Furthermore, take note of 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); however, even in this case, if requests are finding directory entries, multiple sequential batches may be required.

The findget request may save further round trips when file data have to be read. 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. Once more, there is a single round trip to the server, and we may exploit streaming to aid with latency. Figure 1B shows the effect. With a conventional UNIX-style interaction, further RPCs must still be added to Figure 1C to retrieve file data, although batching protocols (eg, later versions of the NFS) may still retrieve data and metadata in a single batched 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 for file removal errors until all requests have been sent. Moreover, 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.

```
errors:= new list of chan of error
dire:= Find("/sys/sre", "~*.o", "/", "/", 0)
while receive dir from dire {
   // get a handle for the dir's tree
   tree:= treeof (dir)
   // issue a remove and note its status channel
   errc:= tree.Remove(dir["path"] )
   errors = append(errors, errc)
}
// now check for errors for all remove requests
for errc in errors {
   receive err from errc
   if err != nil {
      warn (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 Figure 3B 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.

As an aside, each directory entry is self-describing and includes as an attribute the address for the file in the network (ie, 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

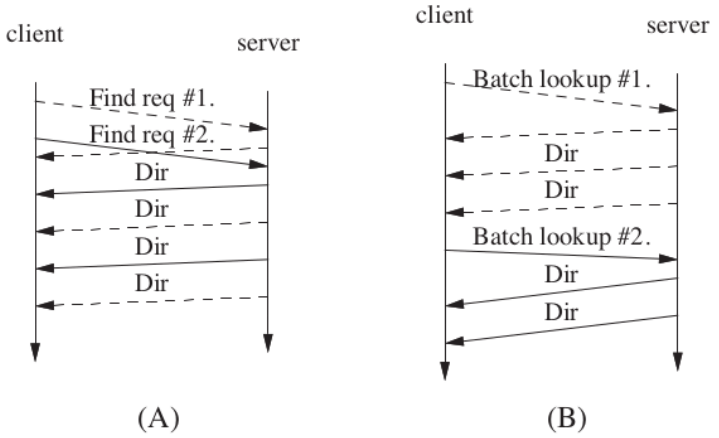

FIGURE 2 In ZX (A), 2 find requests issued may proceed concurrently, and their reply channels may be processed later. In (B), a batching protocol must wait for (batched) replies when traversing a tree to find files because further requests may depend on previous findings

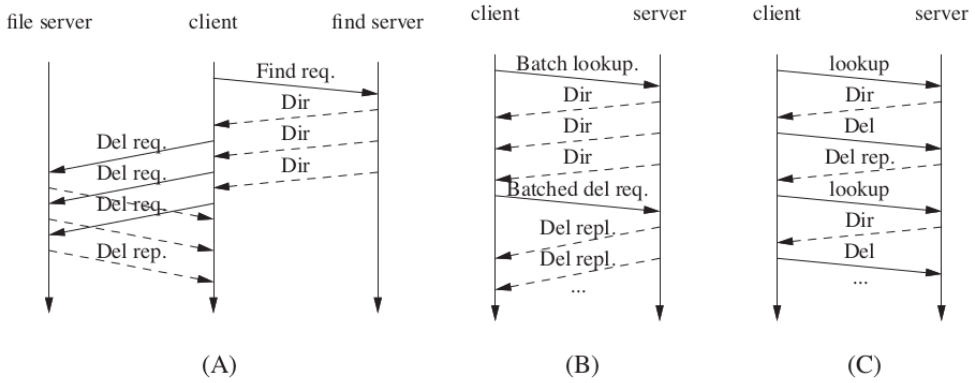

FIGURE 3 Issuing a series of remove requests for files (A) with find, (B) with batched lookup and remove requests, and (C) with classic UNIX-style requests.

	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

2.5 The ZX file system protocol

The ZX protocol follows from the design of the operations shown before. Messages are listed 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, ie, 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 as UNIX would do.

The type field is a single byte indicating the message type. The remaining fields match the arguments of the corre- sponding ZX call. The fields path, topath, pred, spref, and dpref are strings, encoded by sending their length and then their UTF-8 text. The fields off, count, and depth are 8-byte little-endian integers. The 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 are data flowing to (or from) the server due to a request, the data follow 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 stream- ing. 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, as follows.

```
datachan:= Get("/foo", 0, zx.All)
while receive data from datachan {
    err:= process (data)
    if err != nil { // if processing failed
    close datachan with err // don't want more break
}
status:= cerror (datachan)
```

Here, a Tget message is sent through the network. No further data are sent through the request channel. As a reply, file data are 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 are 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).

The code used to update (or create) a file is as follows.

```
datachan:= new chan of bytes with buffering
send filedata to datachan
close datachan
dirchan:= Put("/bar, Dir{"type":"-"}, 0, datachan)
```

```
receive filedir from 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 are 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 2 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 are discarded when it arrives.

When the server refuses to perform the Put request 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. However, it is important to state that it is the message and not the request stream that is 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. However, this does not apply to the data streamed after each Tput message. Such data are written at an offset implied by its Tput request and by previous data streamed in such request. Data written by 2 concurrent put streams would interleave their writes, as UNIX would do. This departs from systems such as CODA [14] which rely on session semantics were full files are updated at a time.

2.6 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). Furthermore, 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 replica models of operation.

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 4 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, such as CODA [14] use session semantics because issuing

1 RPC per write request is unbearable. Systems such as the NFS1 rely on client-side caching and/or batching for multiple requests. However, ZX may just continue stream- ing data while preserving the semantics of its interface. This avoids the need for an extra RPC to the server each time data are being put into the cache, ie, 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 speedup 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 are 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.

2.7 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, 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), 1 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 (1 ms), 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 have 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 are 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 [15].

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 the 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 the Secure Shell File System [16] (SSHFS) 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 that ZX can be adapted correctly for UNIX interfaces and legacy programs.

A Clive Go package, z xux, provides a ZX interface for the 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 entry attributes as expected in Clive. Note that some systems do not implement extended attributes and using them would harm the portability of the tool.

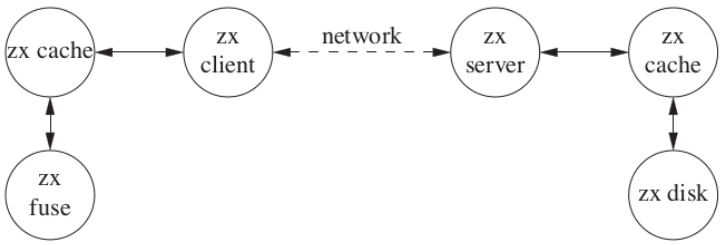

FIGURE 4 A stack of ZX file systems to provide cached FUSE access to a memory-cached server for an on-disk file system

2.8 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 inquire on the status of the

system and may write textual commands into it to issue control requests. Tools such as the UNIX grep command (or Clive's gr) may be used to see who is using the file system, as follows.

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

In the same way, we may see the status of the debug flags and change them using echo, as follows.

```
> 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, as the same idea has been used for decades in Plan 9. However, 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 to use nested control files.

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

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 writen to /Ctl, the server strips the "pass" prefix and issues the resulting update to the next server in the stack. Hence:

```
> 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.

2.9 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 processes 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.

2.10 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 Transmission Control Protocol [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 bro- ken 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 (ie, 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 the NFS) use them. However, one of the caches imple- mented 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 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 timeout.

All this is to say that I/O errors and server failures (crashes) are handled as 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

The main drawback when using ZX is actually a consequence of its design. ZX 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. However, this is the drawback of avoiding a loosely coupled cache for file access. This is also the case when a ZX server goes down due to a power failure at the building or to any other problem.

This problem may be addressed to some extent by hosting the server in the cloud, although we keep our own servers for now. Note that as the so-called CAP theorem (Consistency, Availability, and Partition tolerance [17]) indicates, it is not feasible to obtain both consistency and availability in the presence of network partitions. Because of the problem addressed, ZX favors consistency and sacrifices availability.

We have to say that, to address disconnections, we recently added disconnected operation to one of the ZX caches. While connected, the system operates as described in this paper. Upon network outages, operation continues with the cached data after issuing a warning to the user and asking for permission to do so. 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.

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 were 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, as in the work of Satyanarayanan et al. [14]. In a discarded prototype with session semantics, files were synchronized by client caches when all the data were received. This is fine when different clients do not interact by sharing files. However, updates made by 1 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.

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), if 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 application programming interface (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 the 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 LAN, the NFS might be a better choice.

4 RELATED WORK

There have been 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. 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. To help compare ZX to related works, the following (sub)sections classify other systems according to the technique or feature that we consider as the most relevant for explaining the differences with respect to ZX (although some of the systems might belong to more than one group). Even though for each one of the cited systems there are more differences explained later, in general, we may say that ZX differs from other related systems in the following respects:

- Unlike others, ZX relies on channels and leverages the CSP style of programming.
- ZX includes a find request not found in other systems.

4.1 Networked file systems

First, there have been many NFS protocols successfully used in LANs, since long ago. We can mention the NFS1 and similar systems such as the Common Internet File System [18], the Apple Filing Protocol [19], 9P [20] and Styx [21]. Being designed for LANs, they do not handle high latencies well. On the other hand, ZX does.

The NFS departs from others in this group in that it now considers latency in its design. The 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 the User Datagram Protocol. The latest version, NFSv4, uses the TCP and is no longer stateless. NFSv4 is used by some Cloud Storage providers, such as Amazon's EFS (Elastic FS) [22]. It supports compound RPCs to deal with high latency. The client batches operations and the server batches responses; hence, 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 the 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.

The SSHFS is based on the SSH's Secure File Transport Protocol [23]. Most SSHFS implementations (ie, OpenSSH) are based on SFTP version 3 [24]. The SFTP is based on synchronous RPCs, following the UNIX API for file operations. As a result, SSHFS also requires multiple round trips in many cases. Some efforts are being done to improve its performance by

using windowing [25]. Unlike SSHFS, ZX combines RPCs and streaming by using channels for requests and replies. Furthermore, it has find.

The Low-bandwidth Network File System (LBFS) is a distributed file system designed for low bandwidth [26]. Its techniques are complementary to our work: LBFS considers a limited bandwidth environment, whereas ZX considers latency instead. The LBFS requires, for example, at least 2 RPCs to read a file not in the cache, whereas ZX does not. This is reasonable for the LBFS because it considers bandwidth as the problem and trades transferring data with extra RPCs.

Op [27,28] is designed to transparently bridge Styx [21] over high-latency links. It is a 9P [20] descendant. Op relies on 2 requests to put and get both data and metadata. The data are 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.

Well-known distributed file systems such as, for example, the Andrew File System [29] and Constant Data Availability (CODA) [14] 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. ZX does not operate on session semantics (like CODA does) and, as a result, provides better coherency.

Zebra [30] 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.

GoogleFS [31] and Lustre [32] separate metadata and data. Instead, ZX separates the interfaces and protocols for metadata access and for data access. However, the implementation is free to decide where to put the metadata and the data as long as the interfaces are implemented. Moreover, ZX focuses on high-latency links and not on using parallelism for high-performance computing in lo latency networks.

4.2 Replicated loosely synchronized trees or caches

VisageFS [33] takes a middle stand between loosely connected trees and full coherency, with 3 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.

Orifs [2] provides a 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, such as Dropbox [6], 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 discussed before than it is to Dropbox and related systems.

Systems such as Git [34] and Mercurial [35] can be used to replicate trees at different machines and to synchronize them manually. Tools such as Rsync [36] 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.

4.3 Streamed and batched transfers

Much work has been conducted on asynchronous and streaming RPCs [37,38] 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 as part of the reply. That is different from asynchronous RPCs used by others. The Chirp [8] system relies on a streaming RPC for its streaming mode of data transfers. ZX differs in that its RPCs are channel based and that 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) [38] 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 loop- ing constructs, and that is a safety issue (as Sivathanu et al. [38] 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. Furthermore, 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 [39] 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). Moreover, 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. The source code for this was not shown in the examples for brevity, but there are many examples in the Clive source code.

Chirp [8] 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. As Thain and Moretti [8] observed, the limitation is imposed by the UNIX file API, thus 2 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, due 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.

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 [40]. ZX differs in that it addresses higher latencies (above 50 ms, for example) and in that using find, it avoids the need to batch RPCs for inspecting the file system. Furthermore, ZX uses streaming RPCs instead of batching them.

4.4 Full file transfers

Here, we compare to systems designed to transfer full files to clients to provide remote access, which differs from the interactive use of a remote file system as that provided by ZX. Some permit retrieving and updating partial files, but they are not designed as a replacement for conventional file systems as used by the operating system.

HTTP, WebDAV [41], 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 [42,43]. 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 (ie, 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 are 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.

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. However, 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 such as NFS and SSHFS are well known and provide a common ground for perfor- mance 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 the 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, 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 the earlier versions of the NFS. In particular, this version of the 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 LAN.

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

5.1 Microbenchmarks

Filebench is a well-known synthetic microbenchmark [44,45]. 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 270 MiB and a total of 1100 files. In short, workload rmw1 reads and creates some files, removing one of them. Workload launch 1 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.

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. Furthermore, 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 kernel caching in the measurements.

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

TABLE 2 Filebench bandwidth for Secure Shell File System (SSHFS), Zxfuse, and Network File System (NFS). Using the loopback, with no extra latency

This is the worst case scenario for ZX: no extra latency and using the UNIX interface for file operations through the FUSE driver. Moreover, Zxfuse has not been optimized (while NFS and SSHFS have been) and makes 2 to 3 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 LAN. Although such a version does not exist (yet), we expect an optimized version of Zxfuse to perform closer to SSHFS. However, 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.

	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 Secure Shell File System (SSHFS), Zxfuse, and Network File System (NFS). Using the loopback, with no extra latency

5.2 Macrobenchmarks

The experiments made measured results for different latencies. In most cases, we measured discrete points in the range of latencies considered. For example, 0 ms (no added latency), 1 ms, 10 ms, 20 ms, etc. We did not measure the performance for each pixel shown in the plots, but the marks in the axis show the measurements made. The smallest and largest values shown were always measured, and the rest of the points are simply useful to show the shape of the increase of the magnitude being measured.

Figure 5A 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. The 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 0 to 50 ms, the execution times for ZX go from 4.2 to 5.2 seconds, the execution times for Zxfuse go from 43.168 to 619.5 seconds, the execution times for NFS go from 1.2 to 1231.3 seconds, and the execution times for SSHFS go from 3.9 to 706.2 seconds.

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 20 ms, ZX presents speedups of more than 2 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 5B 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. Furthermore, 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, the results for ZX and NFS are similar to those shown before. The 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 6A 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 (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.

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 20 ms of RTT, it is 3 times slower than SSHFS. However, note

that even up to 10 ms, 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 of 20 ms 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 that may be processed concurrently (both with other remove requests and with the ongoing find). Each Tremove request is replied with a completion status that 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. The SSHFS heavily relies in its cache for walking, which makes it run fast for this experiment (but not faster than speaking ZX). The 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 6B presents another experiment where files are actually read. It computes differences for 2 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).

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. Furthermore, 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. However, note that in the case of the FUSE adaptor for ZX, it is still the ZX interface that is 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 1 round trip.

The next experiment performs a recursive grep for file contents in a Linux kernel source tree. Figure 7A presents the results when using a cold cache, and Figure 7B shows the results for a warm cache. In both cases, ZX is several orders of magnitude faster than their counterparts, with just 10 ms 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 (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 8. The NFS is faster than ZX with the FUSE driver for latencies under 10 ms, as it could be expected. The reason is that the 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 the 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, the NFS outperforms SSHFS as well. One reason for this is that for writes and file creation, the 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 exper- iments are exactly the same ones shown in Figure 8 (compiling a file tree), Figure 7A (running grep), and Figure 6B (computing differences in a file tree), and the setup used for them is exactly as in the previous ones. However, this time, the experiments measure the times for different ZX file system stacks. In particular, we measure stacks with all combina- tions 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 9A presents the results for compiling the SSHFS source tree, Figure 9B corresponds to the grep experiment with cold caches, and Figure 9C shows results for computing file differences.

As it can be seen, using ZX cache at the client has a significant impact in performance. This is as 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. Furthermore, 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, nearby 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.

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 LAN with low latency, the 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 whereas 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. Moreover, 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.

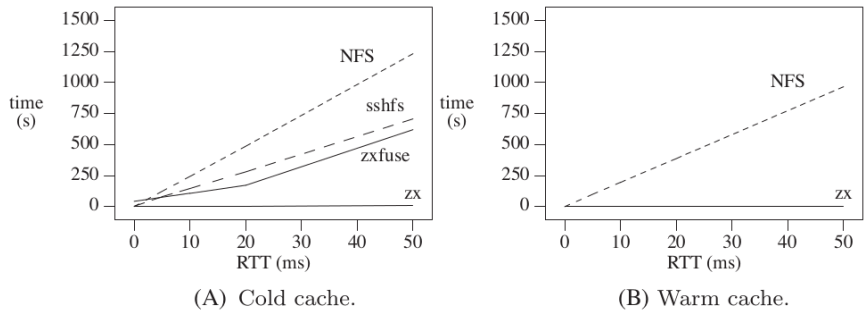

FIGURE 5 Find missing files with no FUSE adaptor as latency increases, for ZX and Network File System (NFS)

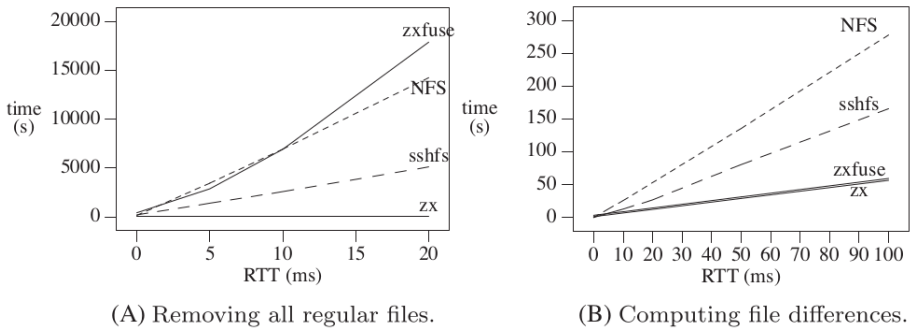

FIGURE 6 Removing all regular files and computing file differences in a small file tree for ZX, ZX through FUSE, Network File System (NFS), and Secure Shell File System (SSHFS) as latency increases

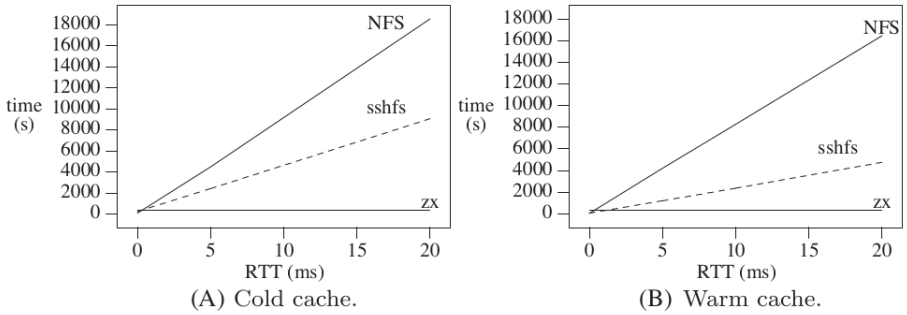

FIGURE 7 Grep for files for ZX, Network File System (NFS), and Secure Shell File System (SSHFS) as latency increases

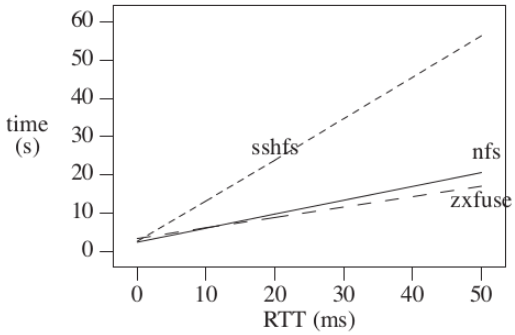

FIGURE 8 Compile the Secure Shell File System (SSHFS) source tree for Network File System (NFS), ZX with FUSE, and SSHFS as latency increases

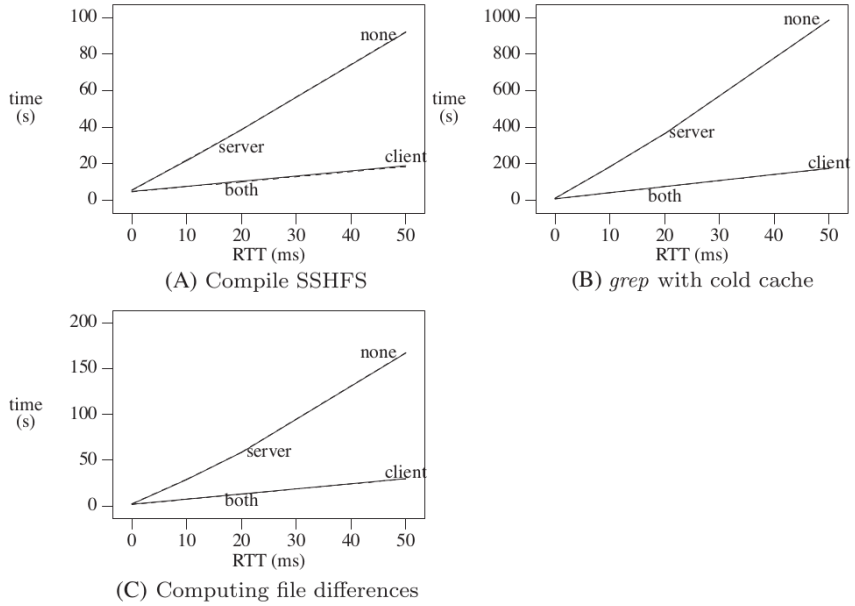

FIGURE 9 Experiments using different ZX file system stack configurations as latency increases. Plots correspond to no caches (none), client cache only (client), server cache only (server), and both client and server caches (both)

6 CONCLUSIONS AND FUTURE WORK

We have shown that, using ZX, a central file server can still be used for interactive workloads despite access through high-latency links with more than 50 ms of RTTs. Evaluation results indicate that using a central coherent server can still be practical in the presence of latency. In fact, while conducting the experiments, other systems measured either caused kernel timeouts when latency was high enough (NFSv4) or were unbearably slow (SSHFS). However, ZX could cope well with latencies measured.

We have shown how the find request prevents many unnecessary round trips and helps perform well when latency is bad.

Examples included have shown how the ZX interface fits well with CSP-like languages (eg, Go), which is important considering that such languages are popular in Cloud and Grid computing applications. The interface provided by ZX is closer to such languages than the one provided by UNIX and related systems (because of channels).

We have also shown that, at a price, ZX may be adapted using FUSE to permit legacy applications to work through ZX.

We have been using ZX for several years now, both on the LAN at work and when using it from home or away. Although the Figures shown indicate that the NFS is a better choice for a LAN (regarding execution times), our experience says that ZX performs well 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 70 ms most of the time, yet using remote files was as convenient for us as it was using a local disk. When used from a metropolitan or a wireless area network, ZX permitted us to continue using a central file server, instead of relying on synchronized (but 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 also plan 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. Shepler S, Callaghan B, Robinson D, et al. Network File System (NFS) version 4 Protocol, 2003. Internet RFC3430.

2. Mashtizadeh AJ, Bittau A, Huang YF, Maziéres D. Replication, history, and grafting in the Ori file system. Paper presented at: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM; 2013; Farminton, Pennsylvania. https://doi.acm.org/10.
1145/2517349.2522721

3. Armstrong J, Virding R, Wikstrom C, Williams M. Concurrent Programming in ERLANG. New Jersey: Prentice Hall Englewood Cliffs; 1993.

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

5. Hoare CAR. Communicating sequential processes. Commun ACM. 1978;21(8):666-677.

6. Dropbox. https://www.dropbox.com

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

8. Thain D, Moretti C. Efficient access to many small files in a filesystem for grid computing. Paper presented at: IEEE/ACM International Workshop on Grid Computing; 2007; Austin, TX.

9. Hedges M, Blanke T, Hasan A. Rule-based curation and preservation of data: a data grid approach using iRODS. Futur Gener Comput Syst. 2009;25(4):446-452.

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

11. Dorward S, Pike R, Presotto DL, Ritchie DM, Trickey H, Winterbottom P. The Inferno operating system. Bell Labs Tech J. 1997;2(1):5-18.

12. Ballesteros FJ. Lsub Go. Lsub TR 15-3, 2015. http://lsub.org/export/golsub.pdf

13. Liskov B, Shrira L. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. Paper presented at: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI'88, ACM, vol. 23; 1988; Atlanta, GA.

14. Satyanarayanan M, Kistler JJ, Kumar P, Okasaki ME, Siegel EH, Steere DC. Coda: a highly available file system for a distributed workstation environment. IEEE Trans Comput. 1990;39(4):447-459.

15. Pike R, Presotto D, Thompson K, Trickey H. Plan 9 from Bell Labs. EUUG Newsletter. 1990;10(3):2-11.

16. SSH Filesystem. http://fuse.sourceforge.net/sshfs.html

17. Gilbert S, Linch N. Brewer's conjecture and the feasibility of consistent, available, partition tolerant web services. SIGACT News. 2002;33(2):51-59.

18. Leach P, Perry D. CIFS: a common Internet System, Microsoft Interactive Developer, November 1996.

19. Sidhu GS, Andrews RF, Oppenheimer AB. Inside AppleTalk. Massachusetts: Addison-Wesley Reading; 1990.

20. Presotto D, Winterbottom P. The Organization of Networks in Plan 9. Plan 9 User's Manual, Vol. 2.

21. Pike R, Ritchie DM. The Styx Architecture for Distributed Systems. Bell Labs Tech J. 1999;5(2):146-152.

22. Amazon Elastic File System. http://aws.amazon.com/efs/

23. SFTP specifications. https://wiki.filezilla-project.org/SFTP_specifications

24. Ylonen T, Lehtinen S. SSH File Transfer Protocol. RFC-Draft, 2002. https://filezilla-project.org/specs/draft-ietf-secsh-filexfer-02.txt

25. Making SFTP transfers fast. http://daniel.haxx.se/blog/2010/12/08/making-sftp-transfers-fast/

26. Muthitacharoen A, Chen B, Mazieres D. A low bandwidth network file system. ACM SOSP. 2001;35(5):174-187.

27. Ballesteros FJ, Soriano E, Guardiola G. Octopus: an upperware based system for building personal pervasive environments. J Syst Software. 2001;85:1637-1649.

28. Ballesteros FJ, Guardiola G, Soriano E, Lalis S. Op: styx batching for High Latency Links. IWP9, 2007.

29. Morris JH, Satyanarayanan M, Conner MH, Howard JH, Rosenthal DS, Smith FD. Andrew: a distributed personal computing environment. Commun ACM. 1986;29:184-201. https://doi.acm.org/10.1145/5666.5671

30. Hartman JH, Ousterhout JK. The zebra striped network file system. ACM Trans Comput Syst. 1995;13:274-310. https://doi.acm.org/10.1145/210126.210131

31. Ghemawat S, Gobioff H, Leung S-T. The Google file system. SIGOPS Oper Syst Rev. 2003;37:29-43. https://doi.acm.org/10.1145/1165389.945450

32. Lustre Filesystem. http://lustre.org

33. Thiebolt F, Ortiz A, Mzoughi A. VisageFS dynamic storage features for wide-area workflows. Paper presented at: Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems; 2007; Cambridge, MA.

34. Git, fast version control. http://git-scm.com

35. Mercurial. http://mercurial.selenic.com

36. Trigdell A, Mackerras P. The rsync Algorithm. Australian National University. http://rsync.samba.org

37. Ananda AL, Tay BH, Koh EK. A survey of asynchronous remote procedure calls. SIGOPS Oper Syst Rev. 1992;26(2):92-109.

38. Sivathanu M, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Evolving RPC for active storage. ACM SIGPLAN Notices. 2002;27(10):264-276.

39. Nightingale EB, Chen PM, Flinn J. Speculative execution in a distributed file system. ACM Trans Comput Syst. 2006;24(4):361-392.

40. Aguilera A, Kluge M, William T, Nagel WE. HPC file systems in wide area networks: understanding the performance of Lustre over WAN. Euro-Par 2012 Parallel Processing. Vol 7484. Berlin Heidelberg: Springer; 2012:65-76. https://doi.org/10.1007/978-3-642-32820-6_9

41. Whitehead EJ, Goland YY. WebDAV: a network protocol for remote collaborative authoring on the Web. Paper presented at: Proceedings of the 6th Conference on Computer Supported Cooperative Work ECSCW99; 1999; Copenhagen, Denmark.

42. Fielding R, Gettys J, Mogul J, et al. Hypertext transfer protocol - HTTP/1.1, 1999. RFC Nb 2616.

43. Nielsen HF, Gettys J, Baird-Smith A, Prud'hommeaux E, Lie HW, Lilley C. Network performance effects of HTTP/1.1, CSS1, and PNG. ACM SIGCOMM Comput Commun Rev. 1997;27(4):155-166.
44. Wilson A. The new and improved FileBench. Paper presented at: Proceedings of 6th USENIX Conference on File and Storage Technologies, USENIX 2008; San Jose, CA.

45. McDougall R, Mauro J. Filebench Tutorial. Sun Microsystems, 2004. https://koala.cs.pub.ro/redmine/attachments/download/603/filebench.pdf