Ballesteros *Journal of Internet Services and Applications* (2017) 8:3
DOI 10.1186/s13174-016-0054-8

Journal of Internet Services and Applications

RESEARCH Open Access

Structured I/O streams in Clive: a toolbox approach for wide area network computing

Francisco J. Ballesteros

Abstract

Most distributed applications and tools used in wide area networks and Cloud computing environments use the UNIX I/O framework. In this framework, processes use file descriptors for standard I/O and file access, with the traditional `open`/`close`/`read`/`write` interface. Although this design has proven to be excellent since the 1970s, it is not appropriate for today wide-area systems because of the implied RPCs and network latency. There are systems relying on message streams that perform well in such environments, but they depart from the toolbox approach embraced by UNIX, making it harder to combine existing programs to solve new tasks. In this paper we describe the design, implementation, and usage of a new I/O framework, built to enable the construction of services in environments with high latency, while preserving the programmability of the system as a whole and making it convenient to combine existing tools and programs. The framework relies on named channels for I/O. Each channel carries a stream of typed data including directory entries, raw bytes, and other application-specific data. Separate commands using the framework may be combined as in UNIX, but still tolerate high latencies as found in distributed and Cloud computing environments, enabling a toolbox approach in such environments.

Keywords: Streams, Input/Output, Operating system, Distributed systems, Cloud computing

1 Introduction

The UNIX design for application development and process I/O has been very successful. Today it is used almost everywhere, including distributed computing applications. However, it was not designed with networking in mind and, in particular, it was not designed for high-latency wide area network links.

One of the benefits of the UNIX approach is scripting, and combining different programs to build new ones. In many cases, administrators and users may combine existing programs to perform their job, without having to write new software. This is called the toolbox approach and it made UNIX very popular. But, due to high latency in wide area networks, this approach becomes hard to use in distributed environments in use today such as those found in most Cloud computing deployments.

For example, using the UNIX `grep` or `find` tools to locate files of interest in a file tree accessible through a WAN link is usually unbearable. In most of the cases, a remote session must be established to the remote system to issue the commands there. But this can be inconvenient. First, access is required for executing software at the remote system, even when the only access desired is for reading files. Second, combining programs that rely on data from more than one system may result in unbearable execution times because executing commands at one of the data sources may lead to a high latency when reaching other data sources (or destinations) involved. We believe that the UNIX I/O model is at the core of this problem, because of the RPCs (Remote Procedure Calls) it implies.

We acknowledge that disk latencies are usually higher than those of the network, as pointed out by others [1], and UNIX copes with them well. But that is not the case when wide-area links are involved. Under high latency, round trip times add up quickly leading to poor execution times. For example, in measurements from [1], reading data from a magnetic disk takes 20 ms and a round-trip from California to the Netherlands takes 150 ms. It is the addition of such round- trips what makes it unbearable to operate on files accessible from a WAN using interactive commands. Asynchronous RPCs and caches have been used to address the issue, but the

Correspondence: nemo@lsub.org
Rey Juan Carlos University of Madrid, Fuenlabrada, Madrid, Spain

Springer Open

© The Author(s). 2017 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

problem still remains in many cases because the interface is not adequate.

Much research has been conducted and other application frameworks and I/O paradigms have been developed to address this kind of environments. For example, Erlang [2] and related systems adopt channel-based communication and a CSP-like style (Communicating Sequential Processes [3]) to leverage streaming for distributed computing applications. Google map/reduce [4] addresses other issues, but also leverages streaming for high performance and departs from the system I/O framework. However, these and related approaches favour writing ad-hoc software for the problem at hand instead of combining existing commands to build new ones.

We have designed and implemented a new system, Clive, for distributed computing environments. Its I/O framework can be used in UNIX (and other systems) to address the problem stated before, which is further described in the next section. That is, it permits using the UNIX toolbox approach even when latencies get high (which is often the case in Cloud computing environments).

In Clive, applications perform I/O through named channels. Named channels do not carry raw byte streams. Instead, they stream typed messages that include directory entries, raw bytes, file addresses, errors, and application defined data. Doing so enables streaming of structured data, avoiding extra RPCs, which permits operation in high latency networks. For example, for file commands, full file trees can be concurrently streamed from one or more servers and pipelines may process such streams without further RPCs to the file servers involved, reducing the effects of latency. Together, named channels and structured streams cooperate to make it practical to combine existing programs to build new ones in distributed environments with high latencies, instead of requiring to write ad-hoc applications in all cases. Existing UNIX programs may be used as well as new ones.

Named channels used for I/O include not just input and output streams, but may include streams for web interfaces and other resources. Because they are named, it is feasible to use their names to create non-linear pipelines from the shell. For example, two different file trees might be streamed from two different servers and compared for differences using a command line. Also, because channels have names, programs may check whether certain I/O channel names are available or not, and change their behaviour accordingly. For example, a program may notice that a channel for web interfaces is available and, in that case, start its web interface and issue through the channel the HTML required to view it, or the URL used to reach the interface.

The contributions of this paper are:

1. A new I/O framework design that performs well on high-latency links.
2. The description of the implementation of the framework for Clive and UNIX using the Go programming language.
3. Examples of use that show how the design actually enables using a toolbox approach despite high latencies.

In the next section we state the problem addressed in this work. Section 3 presents the I/O framework discussed in this paper. Section 4 provides a brief introduction to Clive and to its Go dialect as used in the examples that follow. Section 5 describes the Go I/O framework used in Clive. Section 6 provides examples of file processing commands using it. Sections 7, 8, and 9 introduce other examples for file editing commands, commands using non-linear pipelines, and user interfaces. Sections 10 and 11 describe the implementation and discuss some drawbacks and lessons learned. Sections 12 and 13 provide quantitative evaluation results and discuss related and further work.

2 Problems with UNIX I/O and high network latencies

The problem addressed by this work is that it is not reasonable to require multiple RPCs to reach file and data servers when latency is high, and that using the UNIX toolbox approach requires many more RPCs than needed.

A UNIX style API for I/O relying on the venerable `open/close/read/write` interface requires multiple RPCs to the servers providing the resources being read or written. When the data of interest is being streamed, using read and write to process the stream is fine. However, this interface is neither appropriate to locate the files of interest and feed a stream nor to update (remote) files once a data stream has been processed.

In some cases, using a remote shell to run the desired set of commands at the remote system may suffice. But this is not always desirable and, in some cases, it may be unfeasible. For example, a service provider might be willing to export files or data but may be reluctant to let clients run software or commands at the servers exporting the data. Because of the popularity of Cloud computing environments, it may be hard even to know what “near the data” means. That is, the data might come from a distant server hosted in the Cloud. Furthermore, when more than one system provides the data of interest, and they are far from each other (regarding latency), there is no correct place where to connect to (to execute the commands near the data), because being close to a source implies being away from another.

Also, distributed applications may require non-linear process "pipelines". For example, using multiple processes as sources of data for a single pipeline stage, or using a single data source to feed multiple distributed processes. Conventions in UNIX for using standard input, output, and error streams make it harder to setup such process networks using the system shell. Furthermore, UNIX conventions were appropriate for using text based terminals, but today we have web interfaces, audio I/O, etc. As a result, ad-hoc applications have to be built to glue existing components.

The usage cases addressed by this work are those where users combine different commands to administrate and/or use distributed resources, i.e., when the toolbox approach would be used but is not used because of high network latencies. We admit that not all users rely on the UNIX toolbox approach. In fact, just system and application administrators and, so called, power-users are the ones who prefer to follow it. However, it is desirable to be able to retain such approach (which has worked well for decades) for the distributed environments of today.

3 I/O through structured named channels

The design principle underlying the I/O framework presented in this paper is to avoid the need to reach the data servers to resolve names and/or to iterate through file trees when using a data stream suffices. Following this principle, there are three ideas combined in the I/O framework discussed here to address the problem stated before:

- Enabling data sources to select data of interest and stream the result.
- Using typed messages in streams, to make them capable of streaming structured data and full file trees, and not just linear data.
- Using names for data streams to let users and commands build non-linear pipelines.

The idea is to let programs produce or consume data streams that carry individual messages. Some systems like Oberon [5] use strong typing, while others such as UNIX [6] use raw, untyped, data. We take the middle way: streams sent through channels carry typed data, but application specific data types are unknown to the system. The system forwards messages (as raw data) and it is up to senders and receivers to know what they mean and how to handle them. Messages are typed but there are only a few well-known types: raw bytes, directory entries (file metadata), file addresses (with a file name and a line/rune range), and error indications (encoded as a string with the error message). By convention, messages with raw bytes are considered as the actual data flowing through the stream, which should be processed by commands through the pipeline. All commands usually handle such messages and blindly forward everything else. All other messages provide context for the flowing data. For example, a directory entry followed by raw-bytes messages can be used to send data for a given file, commands similar to grep may send messages with file addresses before sending the matched data, and so on.

The approach is similar in spirit to `roff` pipelines in UNIX. When formatting documents with `roff`, commands are added to the pipeline to format figures, tables, equations, etc. Each command works with the part of the (streamed) document it understands and forwards everything else verbatim. This permits adding new commands and new formatting constructs without disrupting existing tools.

We take the same approach for system-wide I/O. As far as we know, Clive is the first system doing so. Clive commands may cooperate by working on the message types of interest for them and forwarding everything else. The idea is simple but powerful as, hopefully, the following sections illustrate.

Messages carrying application specific data are still uninterpreted and sent as raw bytes by the system, but their type indicates that they are application messages. By convention, commands encode them using JSON, sending a type name as a prefix. They permit programs in a pipeline to send extra information along with the data being processed. As an example, early versions of the framework used such messages to forward file addresses (file names and line numbers) from programs locating lines. Such addresses were consumed by other programs running later in the same pipeline. Since file addresses became often used, they were promoted to a well-known message type, as a convenience. Application specific messages are also used by some text processing filters to select part of the data flowing through the stream by sending all other data as strings. Such ignored data may be promoted to actual data at later stages of the pipeline. There are examples of this in Section 7.

To permit old UNIX programs to work with the new framework, two adaptor programs are used. One converts an input (Clive) message stream into an output (UNIX) raw-byte message stream ready for consumption by UNIX commands. The other does the opposite and converts raw input data into an output message stream. This enables the construction of arbitrary pipelines mixing Clive and UNIX commands.

To feed a stream with data of interest for the task at hand, Clive accepts a `Find` request that selects file data or metadata based on both a file path and a predicate supplied by the caller. This request is further discussed later, but it is important to say that because the I/O framework permits sending typed data, `Find` may stream directory entries, file data, and errors through an I/O stream. The result is that full file trees may be

streamed while avoiding multiple RPCs to the data source.

Message streams are sent through channels that have names. For example, we use "`in`", "`out`", and "`err`" for conventional standard input, output, and error streams. A program may lookup a channel name to obtain either an input or an output channel. Once obtained, the channel may be used to receive or to send data. Other requests permit to bind a channel to a given name as an input or an output channel and to unbind a name.

Using names is important because it permits programs to operate on arbitrary networks of channels and not just on linear pipelines. For example, the Clive `diffs` tool accepts names for streams to be compared for differences. It concurrently receives file trees (one per input channel) and computes and prints file differences. Differences are reported through the output channel and may be further processed by other programs. Unlike in the standard UNIX implementation, there is no need to issue one or more RPCs per file being compared to the file provider (they are streamed).

Names help with other operations as well. The equivalent of a UNIX I/O redirection can be achieved by replacing the channel bound to a given name.

In short, the framework is just (1) named channels with typed messages, (2) streaming processes capable of finding files of interest, and (3) two external adaptors to bridge raw UNIX I/O and Clive I/O. Examples in the following sections show how it can be used in practice.

4 Clive overview

Clive is a system written in Go [7] and made out of services interconnected through channels, which may cross the network. A Clive program may be compiled to run on any platform supporting the Go programming language. Therefore, processes in Clive may run on UNIX (and other systems). In this case we refer to the underlying UNIX system as the host system for Clive. It is also feasible to compile Clive software to run on a experimental native kernel (which is still work in progress). In what follows we refer to the host system as UNIX, but note that any other system supporting the Go programming language may be used instead.

When hosted, Clive is implemented as a runtime library for Clive processes. The library includes a modified Go runtime and a set of Go packages to implement and access Clive services.

The native system is a single address space shared among all processes. The hosted system, which is the one of interest for the purpose of this paper, has an address space per UNIX command, shared among multiple Go processes. Go processes (called goroutines by the creators of the language) are lightweight user level threads multiplexed among multiple UNIX processes. In Clive, each Go process has a Clive process context. A context includes a name space that maps path prefixes to file trees, a path for the current working directory, a set of named I/O channels, and a set of environment variables. When a process is created, it uses the parent's context. But it is also feasible to spawn new processes that have their own contexts.

The system is written using a modified Go compiler that implements the required support for Clive. In the next section we both motivate the need for a modified Go compiler and provide a brief description of the Go language that may be of help to understand the examples shown later.

The central part of the system is a file system exported through a novel file system protocol, ZX [8]. Such protocol is used to export not just regular files, but also services exported as files (as in Plan 9 [9]). File system services in Clive are split into finders and file tree servers. A finder accepts find requests to find directory entries for files of interest. A file tree implements operations on (found) directory entries. This will be seen in file processing examples discussed later.

4.1 Clive's Go

Go is a concurrent language developed by Google and others [7]. The language follows a CSP-like style [3] for concurrent programming, where processes exchange messages to communicate. Unlike in CSP, messages are sent through channels, which are first-class citizens in the language. Channels may be assigned to variables, sent through other channels, etc. Channels can be made both with and without buffering. When used without buffering, senders and receivers synchronize to exchange a message. A select construct waits until a send or receive operation may proceed and then executes it. If more than one can proceed, one is chosen at random.

This code creates a channel for sending integers and sends three of them:

```
c := make(chan int, 3)      // declare c and make it a new chan of int
for i := 0; i < 3; i++ {    // (the chan has room for 3 messages)
      c <- i               // send i through c
}
close(c)
```

The "`:=`" operator both declares and initializes a variable. The arrow is the operator used both for sending (as an infix operator) and receiving (as a prefix operator). The

`close` operation closes a channel so a receiver can know that there is no further data going through it (once all buffered messages have been consumed).

The problem with standard Go is that channels may not be closed by data receivers and that no error indication is reported when a channel is found to be closed, unlike in UNIX pipelines. For example, an error in a process on the middle of a UNIX pipeline propagates both to the left and to the right of the (broken) pipeline. This is necessary to automatically terminate in a clean way other commands involved in the processing of the data stream. In Go, attempting to close a channel used to receive data leads to a panic.

In the Clive's Go compiler, both the sender and the receiver may close a channel, to stop I/O once the buffered messages (if any) have been processed. Also, an error indication may be given to `close`, unlike in standard Go. Such error can be retrieved by calling a new `cerror` primitive. Furthermore, the send operation has been modified to let the sender check whether the send could proceed or not (e.g., when the channel was closed). Unlike in standard Go, closing a closed channel is a no-operation.

The need for channels to behave better with respect to errors (as UNIX pipelines do) justifies modifying the Go compiler and runtime. But there is another reason for doing so. When multiple Clive processes share a single address space, a process exit must close automatically its input and output channels (to let others know). Therefore, it was also necessary to modify the Go runtime to support Clive application contexts including sets of I/O channels that are closed when the last process using a channel as input (or output) exits.

This is the idiom for a process in the middle of a channel pipeline in the Go dialect used for Clive:

```
var inc, outc chan[]byte    // two channels of []byte
      …
for data := range inc {           // while we can recv. from inc
      ndata := modify(data)
      if ok := outc <-ndata; !ok {  // could send ndata to outc, if ok
            close(inc, cerror(outc))
            break
      }
}
close(outc, cerror(inc))
```

The type `[]byte` is a slice of an array of bytes. Although not standard Go, the example shown works fine in Clive and permits processes to behave correctly when errors happen in the middle of pipelines. The code shown breaks the loop upon send or receive errors and the other channel is closed with the error condition of the closed channel. The behaviour is similar to that of a UNIX pipeline in that errors propagate both forward and backward through the splits of the failing pipeline.

Channels may be bridged through a network connection using the `net.Dial` and `net.Serve` calls:

```
func Dial(addr string, tlscfg
…*tls.Config) (ch.Conn, error)
func Serve(addr string, tlscfg
…*tls.Config) (chan ch.Conn, error)
```

The former call returns a pair of "`chan interface{}`" channels in a `Conn` structure, one for sending and one for receiving, and the latter returns a channel used to receive a channel-pair per connection.

The `interface{}` type represents any data. A Go interface is used to hold values that implement a defined set of methods. Any value of a data type implementing the methods defined by an interface may be assigned to a value of the interface data type. The empty interface, `interface{}`, may therefore hold any value of any other type. The language includes type selection (and reflection) constructs that permit retrieving the value held within the `interface{}` value.

What has been said suffices to understand both the interface for the I/O framework and the examples that follow.

5 Named I/O channels in Clive

The Clive I/O framework is an implementation of the I/O framework discussed in Section 3. As a result, it enables the UNIX toolbox approach in high-latency environments. One of the reasons it does so is its novel interface. In this section we describe the interface as provided by the Go implementation for the framework, before proceeding in the rest of the paper with examples of use and evaluation results to support the claims made.

Each process context has its own set of channels, although it may be shared with other processes. This call returns the names for input and output channels:

```
// return two sets of names (for input
chans and output chans).
func Chans() (in []string, out []string)
```

Two related calls return an input or output channel given its name:

```
In(name string) chan interface{}    // return an input chan or nil
Out(name string) chan interface{}   // return an output chan or nil
```

If there is no such channel for input or output, nil is returned. Therefore, it is easy for programs to check out if a channel is available. By convention, there are channels

named “in”, “out”, and “err” similar to UNIX standard input, output, and error streams. But other channels may exist. Clive uses “ink” as a channel to output web interfaces (Ink is the name of Clive’s Web UI framework), and “voice” as the channel to speak to the user.

Note how channels use `interface{}` as the element data type. That is, messages exchanged can be of different data types. Streams sent through channels carry typed data, but application specific data types are unknown to the system. As it was said before, the system forwards messages (as raw data) and it is up to senders and receivers to know what they mean and how to handle them. Messages are preserved and are never split or merged even when sent through the network. The network protocol may issue multiple packets per message but as far as the user is concerned, message delimiters are preserved.

I/O channels may be added to the current I/O set using two other calls:

```
// set an input channel for the given name
func SetIn(name string, c chan
interface{})
// set an output channel for the given
name
func SetOut(name string, c chan
interface{})
```

If an I/O channel with the given name already exists, it is closed before being replaced with the new one.

As an example, this code copies the input for a command to its output:

```
inc := In("in")          // get the input channel
outc := Out("out")       // get the output channel
for msg := range inc {   // receive messages and…
      // try to send them
      if ok := outc <-msg; !ok {
            // if we couldn't close the input w/ error
            close(inc, cerror(outc))
            break
      }
}
// close the output w/ any error from the input
close(outc, cerror(inc))
```

Within a single address space, messages are sent as expected by any Go program through a `chan interface{}` channel. When a message crosses the address space boundary and is sent through an external device (a pipe or a network connection), it is marshalled and written by its sender and unmarshalled by its receiver using a system-wide network format: the message size (in bytes) is written first, the encoded message type follows, and actual message bytes are written last.

As of today, defined message types are raw bytes (`[]byte`), errors, directory entries (`Dir`), file addresses (`Addr`, with a file name and a line/rune range), and user defined types (with application specific data). By convention, `[]byte` messages are considered as the actual data flowing through the stream, which should be processed by commands in a pipeline. All commands usually handle such messages and blindly forward everything else. All other messages provide context for the flowing data. For example, a `Dir` followed by `[]byte` messages is used to send data for a given file, commands similar to grep send `Addr` messages with file addresses before sending any matched data, etc.

6 Using files

In this section we show how the Clive’s approach for I/O, along with a file interface designed for it, can be used in practice. Note how the resulting interactions with file servers involved differ from the ones implied by a UNIX style API.

Clive relies on two different interfaces for using files: finders and file tree servers. There are user-level UNIX implementations for both, to export UNIX file trees to the network.

The following operation provided by finders streams directory entries matching a find request:

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

The request takes a path and a predicate used (at the server) to select the entries of interest. Remaining parameters are not relevant for this discussion, they permit a correct evaluation of the predicate when using name spaces that change paths as seen by the user. As it can be seen, the return value is a channel used to receive directory entries (`Dir` values). Because the I/O framework permits streaming of `Dir` structures directly through an I/O channel, we can use the output channel as I/O for commands.

For example, this command lists all directories under a given path (“>” is the shell prompt):

```
> lf /zx/sys/src,d | pf
d rwxr-xr-x      0 /zx/sys/src
d rwxr-xr-x      0 /zx/sys/src/clive
…
```

The command `lf` produces a stream of `Dir` messages and `pf` is used to print them. The argument given to `lf` asks it to call find to ask the file server for directory

entries for files under "/zx/sys/src" with a predicate "d" (which is a shorthand for "type=d", or "file type is directory").

Figure 1 depicts I/O interactions in this example. Once lf issues a find request, all directory entries are streamed as Dir messages through the output channel to pf. Because the framework and the programmer's interface are designed for streaming, it is still feasible to connect different commands through networks with high latency. Although the example is a command run at a single machine, we could run each process at a different machine if so desired. Moreover, the file tree being listed may come from a remote server implementing find. It is interesting to note that with the UNIX I/O model it would be necessary to issue, at least, one RPC per directory.

The implementation for lf is:

```
dirc := ns.Find(name, pred, "/", "/", 0)  // ask for dir entries
outc := cmd.Out("out")                    // get the output chan
for d := range dirc {                     // receive dir entries
      if ok := outc <- d; !ok {           // and forward them
            close(dirc, cerror(outc))
      }
}
close(outc, cerror(dirc))
```

All it has to do is to forward Dir messages through its standard out channel. Should there be errors finding the files, error messages would be forwarded as well. Unlike in other I/O frameworks, errors will still be synchronized with respect to the rest of the output stream. Of course, the program may also issue diagnostics to its standard err channel, as a convenience. But we defer that as a task for pf. If there's a fatal error it is usually reported by closing the channel with an error indication, and lf forwards that indication as well.

As the source shows, if lf has problems sending its output, it closes its input channel to cancel the ongoing find request. In this case, the server and the system might have performed more work than required, because it continues streaming directory entries until detecting that the channel has been closed. But, on the normal case when the output is still wanted, streaming decreases the effect of latency in network I/O.

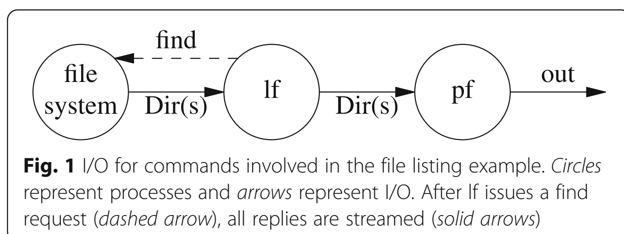

Fig. 1 I/O for commands involved in the file listing example. *Circles* represent processes and *arrows* represent I/O. After lf issues a find request (*dashed arrow*), all replies are streamed (*solid arrows*)

For producing a long listing we can use the "-l" flag for pf; to print just the path names we can use its "-n" flag, and so on. The important point here is that

- lf may stream directory entries and does not require to include the code for printing them in a myriad of different ways, and
- pf may process directory entries as they are being streamed.

A sketch of the implementation for pf can be:

The actual implementation is more complex to handle errors and print messages in several formats according to

```
inc := cmd.In("in")     // get the input channel

outc := cmd.Out("out")  // get the output channel

for msg := range inc {

      // do one thing or another depending on the message type

      switch msg := msg.(type) {

      case error:

            cmd.Warn("%s", msg)     // issue warnings for errors

      case zx.Dir:

            cmd.Printf("%s\n", msg) // print directory entries

      case []byte:

            cmd.Printf("%s\n", string(msg))     // print file data

      default:

            // forward all messages we don't care about

            outc <- msg

      }

}
```

command flags. It can be seen how the program may select messages depending on their types, and also forward all unknown messages along with the program output.

The example becomes more interesting if we modify it slightly to remove all object files:

```
> lf /zx/sys/src/,~*.o | rmf
```

Like before, lf streams directory entries (the predicate now asks file names to match "*.o"). Unlike before, rmf receives them and removes the corresponding files. While removing, the server may be streaming more entries to lf. The remove command is careful to remove path suffixes before prefixes, to permit removing full trees. Therefore, a recursive remove is just

```
> lf /zx/sys/src, | rmf
```

because the empty predicate (no text after the ",") matches all files. Note how the I/O framework permits to combine different commands in different ways while being able to stream data from one to another. The net effect is that many RPCs are avoided. We are using files as the running example, but the same ideas apply to other problems. In the next example we rely on `FindGet`, a request that behaves as find but streams both data (for regular files) and metadata:

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

The resulting stream is a series of `Dir` messages, one per file. Each `Dir` for a regular file is followed by a series of `[]byte` messages containing the data for the file (as shown in Fig. 2).

Now we can print the contents of all Go source files in a file tree with this command:

```
> gf /zx/sys/src,~*.go | pf
```

Here, `gf` calls `FindGet` to stream both matching directory entries and file data for regular files with names matching "`*.go`". The command pf prints each directory entry received, as well as "`[]byte`" messages. Note how the I/O framework permits streaming a full file tree through the pipeline. The resulting stream looks like the one shown in Fig. 2. The I/O interaction is exactly as depicted in Fig. 1 using `FindGet` instead of `Find`. The equivalent UNIX commands would need multiple RPCs to locate files with the desired names and even more RPCs to read those files and print their contents.

Because channels are named, `pf` can simply replace its input stream to accept optional command line arguments for file names. For example, the `pf` code can be kept as-is and it suffices to add this before the code shown:

```
if len(args) != 0 {

        cmd.SetIn("in", cmd.Files(args…))

}
```

The conditional checks for arguments and, when present, replaces the input channel with one for the named files. The call to Files returns a channel streaming file data for command arguments (by calling `FindGet` for each argument); the call to `SetIn` replaces the named input channel with a new one. When arguments are given, the standard input channel gets replaced and streams files for the arguments, without having to write separate functions or adaptors to cover both cases.

Note that although a full file tree is being streamed, we could still use it as input for `rmf` to remove all the files in the tree. The reason is that `rmf` works on `Dir` messages received (using the paths in the directory entries to remove the files) and ignores all other message types (but for errors). The result would be inefficient but it would still work. That is to say that it is feasible to compose different tools in different (perhaps unexpected) ways.

Another example is a command to copy a full file tree to a different location:

```
> gf /zx/sys/src, | pf -o /dst –w
```

Here, "`-o /dst`" asks pf to prefix "`/dst`" to paths received in `Dir` messages and "`-w`" asks `pf` to create the corresponding files instead of printing them. File data received after directory entries for regular files is just sent to the destination files. The implementation uses a put request that accepts an input channel for file data. Therefore, data is streamed to the destination files. But this requires a file server that accepts such request (the Clive ZX file system does) and it is out of scope for this paper.

As another example, we might move the tree to a different device by copying it and removing its old location. Although it is not sensible to remove the files before knowing that they arrived safely at the destination, the example is illustrative:

```
> gf /zx/sys/src, | rmf -o | pf -o /dst –w
```

Because `rmf` forwards input messages (if `-o` is given) we can combine it with `pf` to copy the input besides removing it.

Using messages for actual I/O also helped in other ways. For example, we can loop through lines for an input file with the `ql` shell command:

```
> gf foo.go | lines | for x {echo line is
$x}
```

Here, the shell for construct receives messages and iterates through them. As message boundaries are preserved by the I/O framework, commands may parse the input and produce output at the desired granularity. As

Fig. 2 Example of data streamed through a Clive I/O channel, which can stream a file tree using typed messages. For regular files, file data follows a directory entry. *Boxes* represent messages

a result, parsing for input data can be factored out into separate commands and reused by using its output as an input stream for further commands. For example, commands to split input messages into lines, words, Go function data definitions in source files, etc. are included in Clive.

The availability of such parsing commands makes it easier to write new tools working on their parsed output because they do not need to parse the input by themselves. This would be hard to do with UNIX I/O, because UNIX does not preserve message delimiters in streams (i.e., UNIX may split and coalesce data from different writes when such data is read from the resulting stream).

7 File editing

In this section we provide more examples that show how typed messages in the Clive I/O framework permit programs to cooperate by using different message types. The merit of the approach is mostly how it favours the composition of tools and how it preserves the programmability of the system as a whole. The efficiency gained when streaming helps with network latency is a nice property but, in our opinion, it is not what matters most.

In editors such as Sam [10] a command language is provided to let the user write editing commands. In Clive, the `ix` editor does not include a command language, but uses plain system commands. This can be done because the I/O framework discussed in this paper permits data to have structure (although the system simply forwards messages without interpreting them, like UNIX would do.)

The following example edits all the Go source files in the tree at "`/src`" to put in uppercase the declarations for global variables and structures. The whole picture is shown in Fig. 3, which depicts the streams as seen at each point of the pipeline.

```
> gf /src,~*.go | gr -xf '^(struct|var)'  '^(}|\))\n' |
     trex -c | pf -wi
```

Here, `gf` calls `FindGet` to stream matching files (those under `/src` with names terminating in "`.go`") through its out channel. Directory entries and file data for the matching files are streamed to the output. The resulting output stream might be similar to the one depicted at the top of Fig. 3.

The next command, `gr`, filters the stream so that text between a match for the first expression, and a match for the second expression, is sent as `[]byte` messages; all other text is sent as ignored data. In this example, the two expressions given match the start and the end of a global definition for a variable or structure in the Clive Go compiler. Therefore, the output stream behaves as a selection of parts of the input stream that correspond to such definitions. The second stream in Fig. 3 corresponds to an example output stream at this point.

The next command, `trex`, translates to uppercase all input data when `-c` is given. This changes the raw-data sent through the stream but preserves other message types, as depicted in the third stream in Fig. 3.

Finally, the `pf` command (with flags "`-wi`") writes the data back, including ignored data. The first stream (above in the figure) is the one produced by `gf`. The output resulting from the pipeline is the stream depicted at the bottom of Fig. 3.

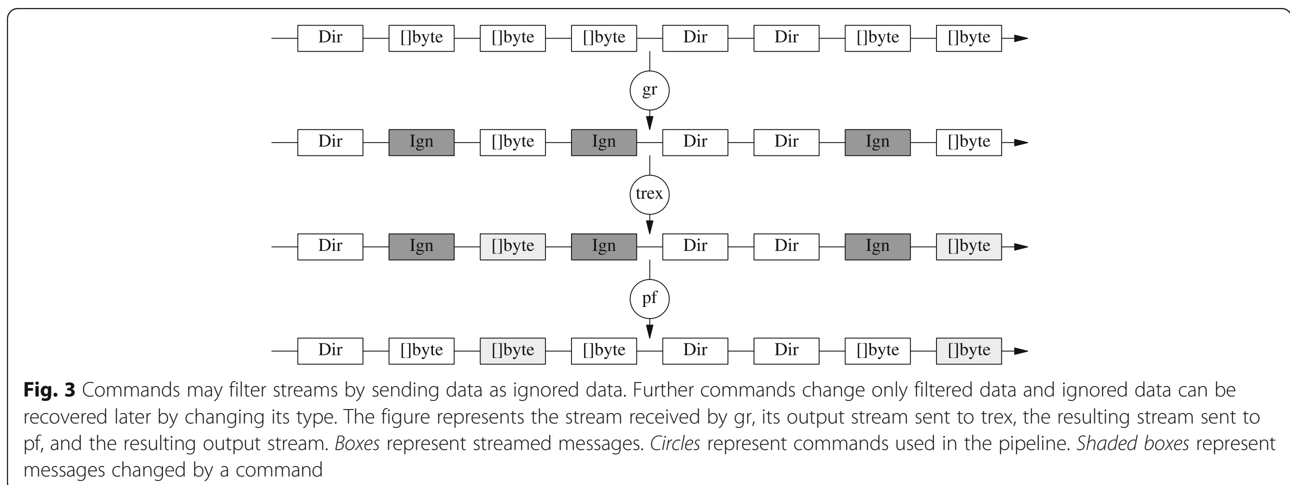

Fig. 3 Commands may filter streams by sending data as ignored data. Further commands change only filtered data and ignored data can be recovered later by changing its type. The figure represents the stream received by gr, its output stream sent to trex, the resulting stream sent to pf, and the resulting output stream. *Boxes* represent streamed messages. *Circles* represent commands used in the pipeline. *Shaded boxes* represent messages changed by a command

The next example does the same, but only for those declarations that include the "ix" string:

```
> gf /src,~*.go | gr -xf '^(struct|var)'  '^(}|\))\n' |
       gr -xfe ix | trex -c | pf -iw
```

The second gr process makes "[]byte" messages not containing "ix" to be forwarded as ignored data, and thus trex ignores them.

The point made by the examples is how using typed messages through channel streams, and making independent programs process those they understand and forward those they do not, enables programs to cooperate in interesting ways. By using "[]byte" messages for data, by convention, and using a few well-known message types, we can get the best of both worlds (typed and untyped I/O systems). Also, because data is being streamed, the result has potential for being efficient; specially over networks with poor latency.

8 Non-linear pipes

When multiple data sources are involved, it is important to be able to build non-linear pipelines to favour a toolbox approach. In this section we present an example using a non-linear pipeline. The example computes differences for files in two file trees. The command

```
> diffs /zx/sys/src,~*.go /other/
src,~*.go
```

takes both arguments and, for each one, creates an input channel streaming the requested files. Each one of the streams is similar to the one shown in Fig. 2. As files are being received, diffs computes and prints the differences. At this point, there is no need to issue further RPCs to servers providing each one of the file trees. Note that the diff tool in UNIX would require multiple RPCs while comparing the input files.

The servers guarantee that for each directory the files are sent in order, sorted by name. By paying attention to Dir messages, diffs may notice if a file is missing in one of the trees or is present in both ones and, in this case, it can receive []byte messages from both streams until the next Dir to learn the contents of the files being compared.

In the previous example, diffs issues a FindGet request to create an input channel for each argument. But the I/O framework permits to use arbitrary pairs of input channels. For example, the next command compares differences for structure definitions in Clive's Go source files:

```
> diffs <{gf /zx/sys/src,~*.go | gr -xf '^struct'  '^}\n'} \
      <{gf /other/src,~*.go | gr -xf '^struct'  '^}\n'}
```

The shell runs each command enclosed in "<{...}" and replaces such part of the command line with the name for a channel to read its output. The command diffs receives two arguments, one per channel streaming file trees to be compared.

This can be done because channels have names and, therefore, their names may be supplied as arguments for commands. The diffs command calls cmd.Files (also used in Section 6) to obtain an input channel for each argument. This function knows how to handle special names that refer to channels instead of files. When the argument starts with the pipe character, what follows is the name for an input channel, and cmd.Files simply returns such channel instead of issuing a FindGet request.

It is interesting to note how besides the composition capabilities of the Clive I/O framework, there is potential for efficient execution. In this example, the file server is in charge of locating and streaming file contents for both trees. As the streams are being received, the command computes and streams differences. There are inefficiencies as well: if a file is present in just one tree, it is still retrieved by the command, although all its contents will be discarded.

9 User interfaces

Named channels permit composing programs in new ways, and not just as part of a pipeline. In this section we discuss a few examples of user interfaces that rely on such feature. The point being made is how the design of the I/O framework enables different tool types to be combined in new and interesting ways. This is important in distributed computing environments where multiple kind of services are the common case.

We wrote this paper using ix, a Clive editor (and shell) that relies on a Web browser as the user interface (see Fig. 4).

The gr command used in previous examples issues an Addr message with a file address (including a line range and a rune rage) for each matching text received.

In the previous examples, pf discarded such messages for printing because it was not asked to print them. But, running gr from the ix editor produces a different effect, because ix pays attention to such messages.

For example, consider this command to grep for function headers in all Go source files within the current directory:

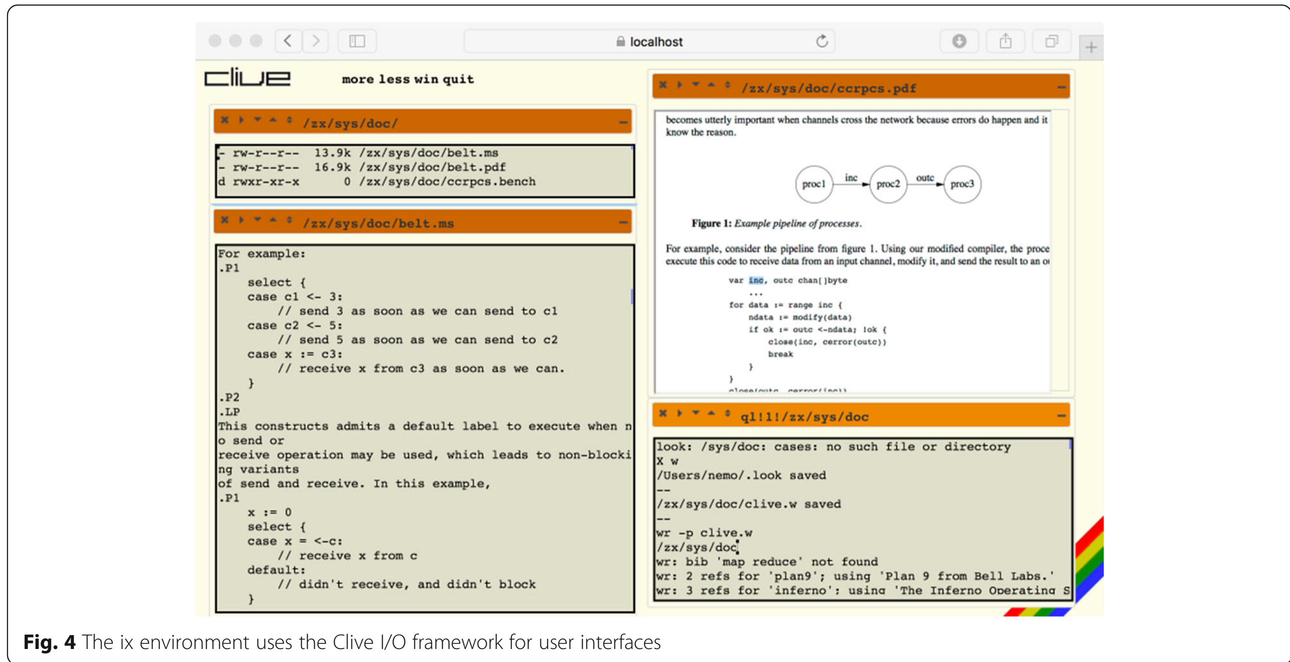

Fig. 4 The ix environment uses the Clive I/O framework for user interfaces

```
> gf.,~*.go,1 | gr -xf'^func'
```

When running in `ix`, the editor opens the matching file(s) in the addresses received and, for each file, it selects the first range found. The mouse can then be used to iterate through the address list. The interesting point here is that `Addr` messages are used to let `gr` and `ix` cooperate, but other commands may remain unaware of them and continue working as expected.

Another example of how named channels permit program composition is the package for Web user interfaces, Ink. By convention, the `ink` output channel is used to output HTML and URLs for user interface elements. A program may check if there is such channel, and provide a user interface in that case, using code like

```
if inkc := cmd.Out("ink"); inkc != nil { // if there's an "ink" chan
        startUI(inkc)        // output a user interface through it.
}
```

The same command might use its standard output when no such channel is available. Programs with a stand-alone Web interface run a Web server and output their URL through the `ink` channel if available, or print it otherwise to let the user know where the interface is.

The `ix` environment creates an `ink` channel for each command it runs and pays attention to messages received from it. This is enabled by the design of the I/O framework. For example, this command opens the Lsub web page within `ix`:

```
> eco http://lsub.org >[out:ink]
```

The redirection “`>[out:ink]`” makes the shell call `SetOut` to replace the `out` output channel with the `ink` output channel. When `eco` echoes its arguments, they are sent through `ink`. Then, when `ix` receives the message and notices it contains a URL, it opens the web page.

UI controls provided by ink use websockets [11] and provide a `WriteTo` method to output the HTML required for them. Writing them through the `ink` channel suffices to make the controls available for the viewer used. Usually, the HTML for the control dials the ink web server through a websocket. Then, the control implementation in the server forwards events and updates to keep the multiple views of the control synchronized.

10 Implementation

Clive is implemented as a set of packages for the Go programming language and a modified Go compiler and runtime system. As of today it runs on systems such as Linux and OS X, but there is an ongoing effort to build a bare-hardware runtime system enabling Clive applications to run as a kernel. For brevity, we describe here the parts relevant for the purpose of this paper. A full description of the changes and additions made to Go is available elsewhere [12].

The Go run-time system has been changed to modify error handling in channels. As described early in this paper, Clive channels can be closed with an error indication and may be closed both by senders and receivers. This suffices to let the programs use channels that are

bridged through external devices like pipes and network connections. A few primitives were added to the runtime to support sending with a boolean return status, calling `close` with an optional error argument, and retrieving the error status for a channel. Also, the implementation for send, receive, and select had to change to disregard sends and receives on closed channels, without panicking, and returning an error in the case of sends.

Another central part of the implementation adds Clive application contexts to the runtime system. Each Go process (goroutine) carries a Clive application id inherited from the parent process. A new call makes a process become the initial process of a new context, detaching it from its old context. This call is used to create a new context to run a given function in a new process. Another call returns the context identifier for the caller process.

The `cmd` package contains the data structures representing resources for a Clive context: I/O sets, name space, environment variables, and current working directory. To play safely with the garbage collector, this is implemented outside the runtime system as a set of resources hashed on the Clive context identifier.

When a process is created, it shares the context with the parent (both processes use the same I/O sets, name space, etc.). But it is also feasible to use `cmd.New` to spawn a new process that has its own context and runs a given function.

```
// create a context to run fun and return
it.
// If wc is given, fun won't run until wc
is closed.
func New(fun func(), wc …chan bool) *Ctx
// Return the current command application
context.
func AppCtx() *Ctx
```

The call permits the caller to customize the new context before the function runs: each resource may be cloned to use a separate copy of it, may be shared with the parent context, or may be started as a fresh (empty) new instance. This process architecture is similar to the one used in systems like Inferno [13], but for the I/O framework shown in this paper.

Resources like environment variables are initialized from the values found in the underlying UNIX when a new UNIX process with Clive code runs. Thereafter, calls like `cmd.SetEnv` modify the values in the Clive context. When a Clive program runs an external (UNIX) process, its environment is carefully initialized to reflect the values from the parent Clive context. This permits the integration of Clive processes both within and outside a UNIX context.

The name space and the current working directory are defined by environment variables in Clive. The name space is a textual description of a table mapping path prefixes to remote directories mounted on them. For example,

```
NS='/ /
        /zx   tcp!nautilus!zx
'
```

makes the root of the underlying UNIX available at "`/`" and the file tree from nautilus available at "`/zx`".

When used within a single UNIX process, the implementation for Clive channels in the runtime and a per-context table of input and output channels (indexed by their names) suffice. The implementation for making them work across different UNIX processes is more elaborate as we describe next.

When running hosted on UNIX (or other systems supporting Go), a Clive channel must use an underlying UNIX file descriptor to send or receive messages to or from the outside world. The implementation combines both file descriptors and environment variables to export channels to the outside of the UNIX process, and to import them.

At initialization time, the Clive runtime scans the set of environment variables for names representing open Clive channels. The value of such variables indicate whether a channel is for input or output, and which open UNIX file descriptor is to be used for reading or writing messages.

Initially, UNIX standard input, output, and error streams are converted into Clive `in`, `out`, and `err` channels. Descriptors are 0, 1, and 2 as expected, and variables are defined to represent them.

When a channel is used between several UNIX processes, it is implemented as a UNIX pipe. Usually, a Clive process creates a channel for input or output and adds it to a new context running the new UNIX process. When the runtime is asked to start the external process, it creates the required pipelines for the new channels, and leaves the descriptors for reading or writing them open at the child process. Environment variables are defined to tell the runtime of the child which descriptors are available for input channels and which ones are for output channels. When the child starts, it looks at the environment variables and, for each channel, creates a Go channel along with a reader or writer process to read messages and send them through the channel or to receive messages from the channel and write them to the UNIX file descriptor. An important detail is that such

reader/writer processes are not started until the application calls `cmd.In` or `cmd.Out` to ask for an input or output channel. This solves an interesting problem: if an input stream was read before knowing that the application wants to receive from it, some data would be consumed, and perhaps lost if the process exits without receiving. Instead, if a Clive process do wants to receive from an input channel, it calls `cmd.In` to retrieve its input channel and, at that point, the reader process is started.

The prototype for the native system is easier to implement in this respect, because it is a single Go address space and there is no problem in sharing channels or other resources in the single address space.

For hosted environments, the implementation works as discussed in this section. The Clive shell, `ql`, relies on the `cmd` package and, for non-builtin commands, arranges for channels to connect external processes as described here.

11 Drawbacks and lessons learned

Combining UNIX and Clive commands requires extra processes to bridge raw UNIX byte streams to messages and vice-versa. The pf command knows how to write messages as raw bytes suitable for UNIX terminals and commands. Another command, `rf`, reads bytes from its input and sends messages ready for Clive commands to consume. Also, many Clive commands have a flag to use UNIX output (raw bytes) without requiring a pipe to `pf`.

Using a hosted environment implies overhead for reading and writing messages instead of raw bytes, and in some cases requires extra processes such as `rf`. However, this permits leveraging existing UNIX tools.

One of the advantages pursued for native environments is avoiding data copies, by sending pointers to data through channels in a shared address space. But we learned using the hosted environment that, in many cases, it is desirable to actually copy the data before sending it to the another process. The reason is that the sender usually modifies the data received before sending it, and making a copy is necessary to avoid race conditions. Also, copying is necessary on hosted environments because data goes through external (software) devices provided by the underlying system.

Before using the approach described in this paper we used `[]byte` as the data type exchanged through channels, following the UNIX I/O design. This worked well but permitted sending only a single error at the end of the data streams (as the close condition for the channel), and required manual encoding for streaming full trees as in `FindGet`.

In the end, many commands were encoding and decoding popular message types over raw `[]byte` channels, and it was more convenient to move the encoding into the system, by making the channels transmit different data types, including both `Dir` and `[]byte`. The system now streams raw bytes including a message type and message length along with each message. Using this, streaming `[]byte` messages is efficient and easy. Also, by adding to the Clive library encoding for a few types used in many commands, it is feasible to send messages for directory entries, errors, and a few others.

For the cases when applications need to send their own types, the `Ign` message type includes as data an application defined type id, and the message bytes. That is enough to let such applications encode and decode the actual messages. This is not complex because any type implementing methods to encode and decode can be sent as-is, and the Clive library will rely on such methods to do the actual encoding and decoding.

Last but not least, perhaps the main drawback of the proposed framework is that it requires writing new programs to be able to use it, because existing tools may be bridged as discussed but still use the UNIX-style API. For example, we had to write `gf`, `diffs`, and other programs to actually use the framework in high-latency environments.

12 Evaluation

To the best of our knowledge, the I/O framework shown here presents a qualitative difference with respect to other I/O frameworks, including the one in UNIX. Evaluation results shown here try to justify that this framework can be used in practice and that it can be efficient. But, in our opinion, what matters most is the composition and programmability of system programs hopefully illustrated in previous examples.

As a micro-benchmark, Fig. 5 depicts the times for transferring a single message through an external device (an operating system pipe) using raw read and writes (solid line), using channels of `[]byte` bridged through

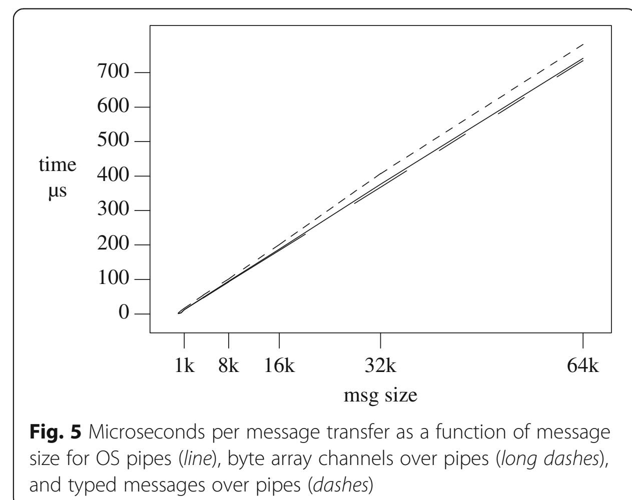

Fig. 5 Microseconds per message transfer as a function of message size for OS pipes (*line*), byte array channels over pipes (*long dashes*), and typed messages over pipes (*dashes*)

the pipe (dashed line with long dashes), and using channels of `interface{}`.

Using raw `[]byte` channels is as efficient as using the raw external pipe. The worst times are for channels of `interface{}`, because using them requires extra time to retrieve the values behind the `interface{}` messages exchanged and extra time to encode the message types and to decode them. In any case, using `interface{}` seems to be efficient enough to be used when it is convenient to transfer multiple data types and not just raw bytes.

Figure 6 shows the total time to perform a recursive `diff` for two copies of Linux's `src/crypto` directory, one of them with an extra byte appended to all files. The lines shown correspond to the UNIX `diff` command using NFS and to Clive's `diffs` command using Clive's I/O.

It is not fair to compare UNIX and Clive when latency is high because UNIX was not designed with network latency in mind. As latency increases, the round trip times required for each RPC add up quickly. However, the Clive I/O framework is designed for streaming.

The point made in this experiment is that Clive's I/O framework, because of its streaming capabilities, can outperform that of UNIX when the operations made can leverage streaming. But it should be noted that any other streaming I/O framework will exhibit similar speedups. Also, for very low latencies, there is overhead paid to convert a raw byte stream to and from messages, as shown in the previous micro-benchmark.

When latency is close to zero, the overhead for Clive's I/O is noticeable with respect to UNIX. For example, the time to grep for "`Clive`" in the source for all Clive commands is 22 milliseconds using a recursive grep in UNIX. In the same environment, hosted, the command

```
>gf src/clive/cmd,- | gr -u Clive
```

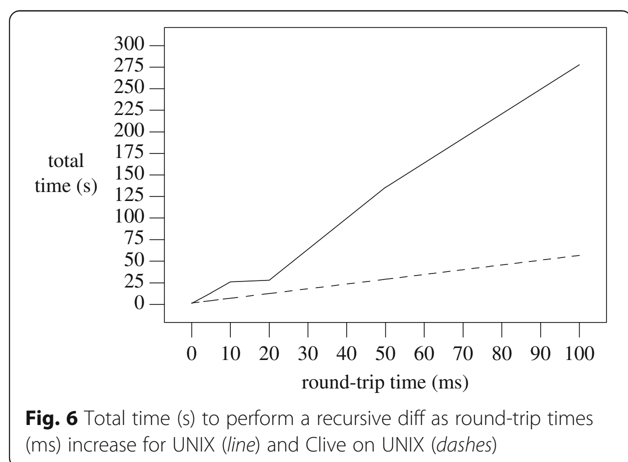

Fig. 6 Total time (s) to perform a recursive diff as round-trip times (ms) increase for UNIX (*line*) and Clive on UNIX (*dashes*)

performs the same task and takes 88 milliseconds. In this case, we are paying in both `gf` and `gr` the overhead of making a Clive's file interface for an underlying UNIX file system, and converting raw bytes to messages and vice-versa. Furthermore, the implementation in Go for Clive's commands is not optimized and relies on multiple Go processes, while the native `diff` command is a fine-tuned, sequential, C implementation. Because the command was measured within a single machine with no extra latency added, UNIX outperforms Clive.

We argue that times for using the I/O framework in the hosted environment are reasonable and show that the system can be used in practice. In those cases when the latency increases, Clive's I/O may enable commands to outperform those of the underlying system by leveraging streams.

Also, the overhead paid in the hosted system permits using both Clive's and UNIX's shells as well as combining UNIX and Clive tools, which is convenient.

13 Related work

Erlang [2] is a language departing from the UNIX interface for I/O. Like Go [7] it relies on a CSP-like style where processes exchange messages through channels. Unlike Go, it can assign processes to different machines and use Erlang process addresses to deliver messages across the network. The Clive I/O framework presented here also relies on channels, but, unlike in Erlang, separates channels from processes following the Go style of concurrent programming. Furthermore, Clive commands can be used as tools similar to UNIX commands and can be combined with other UNIX commands. Clive is closer in spirit to the UNIX toolbox approach than Erlang is, because it has been designed as a new system and not as a language for distributed applications.

File systems like Chirp [14] have used streaming to address latency issues. Chirp [14] is a file system protocol designed to improve performance when accessing files in Grids. 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 our work, The Clive I/O framework differs in that it does not change the communication mechanism depending on the file or the data size, thanks to channel-based streaming facilities. Besides, the Clive I/O framework is used as a general tool

for inter-process communication, and for accessing external devices, and not just for the file system.

There are many streaming and asynchronous RPC mechanisms. To name one, Google's gRPC [15] is a mechanism for both synchronous and asynchronous RPCs capable of streaming requests and replies. It is a communication mechanism related to the channel-based, streaming, RPC mechanism underlying Clive. TChannel from Uber [16] is another middleware-level RPC-like protocol focusing on building reliable distributed services. The work presented here has a different focus, and provides a system interface for I/O intended to replace the UNIX interface. Clive's I/O is not an RPC mechanism. Nevertheless, although out of scope for this paper, the channel-based RPC mechanism underlying Clive differs from gRPC and TChannel in that it is closer to a CSP style of programming than gRPCs and TChannels are.

There are many micro-kernel based systems using message passing for process I/O [17]. See for example Mach [18] or L4 [19]. In those, the I/O system is designed for fast, machine-wide, inter-process communication. In many cases a UNIX system is built upon them, providing conventional pipes and other UNIX abstractions [20]. Microkernels like QNX [21] use message passing for user-level components like the window system. Clive is not a new kernel architecture, but is designed to interconnect both local and distributed components and tools in a CSP-like framework. The I/O system presented here enables a new framework for structuring user commands as stream producers, consumers, and filters, and therefore addresses a different problem.

Oberon [5] is representative of systems using strongly typed systems to interconnect system components. Unlike in Oberon, Clive's I/O can be extended across the network by relying channel I/O through network connections, and not just through UNIX pipes. Also, Oberon has a single process structure while Clive and its I/O framework are designed for multiple processes.

Middleware systems like MPI [22] were designed to exchange messages across the network for High Performance Computing applications. Unlike in the framework shown here, they are not easy to combine with existing UNIX tools. They require significant changes in existing programs to let them use MPI (or similar frameworks).

Document formatting systems like `roff` [23] use pipelines with different commands to process and filter document streams. They are similar in spirit to the approach used in Clive for processing I/O streams (at least for editing commands). Of course, `roff` like tools rely on conventions to be able to cooperate in a pipe-line more than they rely on the mechanisms provided by the underlying system, and they are specific-purpose tools.

The UNIX Streams [24] framework (and other frameworks deriving from it for later UNIXes) introduced control messages processed by stream modules along with data messages. Thus, they are predecessors for the approach of using different message types to control processing in data streams. The I/O framework presented here is a different design, because it addresses a distributed computing environment and because it departs from the UNIX interface.

Frameworks like Google's map-reduce [4] depart from the venerable UNIX interface to provide a high performance framework for I/O and computing in Grids and Clusters. They provide streaming capabilities for particular application domains and, in those domains, they can easily outperform designs like the one presented here. However, Clive's I/O is a general purpose I/O framework, and therefore addresses a different problem.

The X-kernel [25] introduced a path abstraction to replace processes, focusing on data paths instead of processing. This is similar to Clive's approach in that Clive relies on I/O streams to convey data and permit using different message formats. The main difference is that Clive still preserves the process abstraction to adapt the UNIX toolbox approach to modern distributed scenarios and to interoperate well with existing programs.

14 Conclusions and future work

In this paper we have shown how Clive named channels, carrying structured data streams, can be efficient and effective for use in distributed systems. We have shown how they permit the construction of tools by combining other tools, following the UNIX toolbox approach.

We have also shown how the I/O system can be leveraged to apply commands on filtered data streams. This has been done by `roff` since long ago, but this paper shows how it is a natural result when done in a system-wide manner.

The examples included illustrate the resulting system programmability, when using this framework, and also that the I/O system may stream data without requiring extra complexity in the programs involved.

Evaluation shows how the hosted system has overhead with respect to the underlying UNIX, as it could be expected, but shows that it can perform better when latency is considered.

The I/O framework described in this paper has proven to be flexible and convenient when used by programmers. The Clive system is operational, and it is being used daily. As of today only the hosted system can be used. A native kernel is work in progress, and it is expected to be ready in a few months. The implementation requires profiling and some parts of the system might be rewritten as a result.

Abbreviations
API: Application Programmer's Interface; CSP: Communicating Sequential Processes; HTML: Hyper-Text Markup Language; I/O: Input/Output; JSON: Javascript Object Notation; RPC: Remote Procedure Call; URL: Uniform Resource Locator; WAN: Wide Area Network

Acknowledgements
The author thanks Gorka Guardiola and Enrique Soriano for their help. Gorka also improved some of the Clive commands and measured performance for `diffs`. Both helped with discussions and related work.

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

Availability of data and materials
The manual and the software are available as open source at http://lsub.org/. Being a research system, it is still evolving and it is likely it will continue to do so for some time.

Authors' contributions
The author designed the architecture of the system described in this paper and implemented it.

Authors' information
Prof. Francisco J. Ballesteros (http://lsub.org/who/nemo) got his MS in CS on 1993 and his PhD on CS on 1998, at Technical University of Madrid. While an undergraduate, he got several grants from European research projects where he developed run-time software for programming languages. Later, he worked for several years on telecommunications companies, doing systems software (He is the (co)author of LiS, a STREAMS framework for Linux.) Since 1995 he has been a professor at several Spanish Universities where he has been teaching and developing Operating Systems. He developed the Off++ kernel, for the 2 K Operating System jointly with the SRG at University of Illinois at Urbana Champaign. He has been working in R&D on topics related to Plan 9 from Bell Labs, including the Plan B and Octopus Operating Systems. He is also the author of the Omero and O/live window systems. Currently he is the head of the Systems Lab at Rey Juan Carlos University (http://lsub.org) where the Clive OS for Cloud computing environments is being developed.

Competing interests
The author declares that he has no competing interests.

Received: 13 September 2016 Accepted: 21 December 2016
Published online: 02 February 2017

References
1. Dean J. Designs, lessons and advice from building large distributed systems. 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware. Keynote. 2009.
2. Armstrong J, Virding R, Wikstrom C, Williams M. Concurrent programming in ERLANG. Prentice Hall PTR; 1993.
3. Hoare CAR. Communicating sequential processes. Commun ACM. 1978; 21(8):666–77. New York, NY, USA.
4. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–13.
5. Wirth N, Gutknecht J. Project Oberon. New York: ACM Press; 1992.
6. Ritchie OM, Thompson K. The UNIX time sharing system. Bell System Technical Journal. 1978;57(6):1905–29. Alcatel-Lucent.
7. The Go Programming Language. The Go Authors. http://golang.org. Accessed 2014.
8. Ballesteros FJ. Clive's ZX file systems and name spaces. Lsub-TR/14/2. Also in http://lsub.org/export/zx.pdf. Accessed 2014.
9. Pike R, Presotto D, Thompson K, Trickey H. Plan 9 from Bell Labs. EUUG Newsletter. 1990;10(3):2–11. Autumn.
10. Pike R. The Text Editor Sam. Software, Practice, & Experience. 1987;17(11): 813–45.
11. Fette I, Melnikov A. The websocket protocol. Prentice Hall PTR; 2011.
12. Lsub Go, Ballesteros FJ. Lsub TR15-3. http://lsub.org/export/golsub.pdf. Accessed 2015.
13. Dorward S, Pike R, Presotto DL, Ritchie DM, Trickey H, Winterbottom P. The Inferno Operating System. Bell Labs Technical Journal. 1997;2(1):5–18.
14. Thain D, Moretti C. Efficient access to many small files in a filesystem for grid computing. IEEE/ACM International Workshop on Grid Computing. 2007. p. 243–250
15. Ryan L, et al. gRPC: a high performance, open source, general RPC framework that puts mobile and HTTP/2 first. http://grpc.io. Google. 2015.
16. tChannel. Network multiplexing and framing protocol for RPC. http://uber.github.io/tchannel. Accessed 2015.
17. Hartig H, Hohmuth M, Liedtke J, Wolter J, Schonberg S. The performance of micro-kernel based systems. ACM. 1997;31(5):5–18.
18. Bolosky WJ, Fitzgerald RP, Scott ML. Simple but effective techniques for NUMA memory management. Proc Twelfth ACM Symposium on Operating Systems, Operating Systems Review. 1989;23(5):19–31.
19. Au A, Heiser G. L4 User Manual–Version 1.5. Prentice Hall PTR; 1998.
20. Lackorzynski A, Danisevskis J, Nordholz J, Peter M. Real-time performance of L4Linux. Proceedings of the Thirteenth Real-Time Linux Workshop, Prague, Czech. Vol. 2022. 2011.
21. Qnx Neutrino OS developer support. Qnx. http://www.qnx.com/developers/docs/momentics621docs/momentics. 2004.
22. Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface standard. Parallel Comput. 1996;22(6):789–828. Elsevier.
23. Ossanna JF. NROFF/TROFF user's manual. Prentice Hall PTR; 1976.
24. Ritchie DM. The UNIX System: a stream input output system. ATT Bell Laboratories Technical Journal. 1984;63(8):1897–910. Wiley Online Library.
25. Hutchinson NC, Peterson LL, Abbott MB, O'Malley S. RPC in the x-Kernel: evaluating new design techniques. Proc Twelfth ACM Symposium on Operating Systems, Operating Systems Review. 1989;23(5):91–101.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Immediate publication on acceptance
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com