Network services in Clive

Francisco J. Ballesteros

ABSTRACT

Clive is an operating system we are building written in Go using a CSP style that introduces a new file system interface and related utilities. This paper describes tools used to simplify the construction of network services.

Go channels in Clive

In a previous paper [1] we described how channels are bridged to external entities like pipes and network connections. In short, a duplex connection to the outside world is represented by this data type:

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

This function builds a *Conn* out of a file descriptor for an external pipe or network conenction:

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

The *In* and *Out* channels (like any other channel) may be closed, with an optional error indication, and sending and receiving report the failure when the channel used was closed. The new *cerror* call reports the error for a channel that was closed. This suffices to understand what follows, refer to [1] for further details.

Servers

A server is a process (or more) that accepts connections from clients to provide a service:

```
type Srv struct {
        Once    bool // if true, server exists after one client
        ...
}

func New(name, addr string, h CliHandler) *Srv
func (s *Srv) Serve() error
func (s *Srv) Stop(wait bool)
```

The server is created by calling *new* and then calling *serve* to start its operation. The address given to *new* specified where the server should listen for new calls.

The service is not provided by the server. Instead, any type implementing

```
type CliHandler interface {
        HandleCli(c net.Conn, endc chan bool)
}
```

may provide the service. Each connection leads to a call to handle the new client (in a new process). When the server decides to stop it notifies the handler through *endc*.

In almost all cases, the client handler is built by calling this function:

```
func NewChanHandler() (CliHandler, chan *nchan.Conn)
```

It returns a client handler that can be used to create a new server, and a channel where new client connections will be reported. Each such connection will be a *Conn* structure like the one defined before. From this point on, servers are relieved from handling the network connection directly. Instead, they operate on channel connections to serve clients.

For example, this is an echo server:

```
hs, hc := NewChanHandler()

// This is an echo server from inb to outb.
for h := range hc {
        go func(h *nchan.Conn){
                defer close(h.Out)
                for m := range h.In {
                        if ok := h.Out <- m; !ok {
                                t.Fatalf("send: %v", cerror(h.Out))
                                break
                        }
                        Printf("srv: msg %s0, string(m))
                }
                Printf("srv: done %v0, cerror(h.In))
                close(h.Out, cerror(h.In))
        }(h)
}()
```

To make the server listen for network calls we might just do this:

```
s := New("test", "tcp!*!8081", hs)
if err := s.Serve(); err != nil {
        t.Fatalf("serve: %v", err)
}
```

Clients usually dial the network address and then call the function shown elsewhere [1] to bridge the connection to a pair of channels:

```
c, _ := net.DialTCP(n, nil, taddr)
nc := nchan.NewConn(c, 5, nil, nil)
// send a request...
nc.Out <- []byte{...}
// or receive a reply...
data := <- nc.In
// or check out the error if the chan was closed
Printf("sts is %v", cerror(nc.In))
```

Servers and client might multiplex their connections to issue multiple concurrent streams of requests and replies. See [1] for further details.

Clients and the dial service

Instead of dialing servers directly, clients tend to use the *dial service.* This service is packaged in a set of functions:

```
func NewPipe(name string, maker PipeMaker)
func DelPipe(name string)
func Dial(addr string, cfg *tls.Config) (*nchan.Conn, error)
```

Dial is the important one. It understands address strings and returns a connection (based upon channels) to talk to the service at the given address.

Address strings may be of the form

```
network!machine!service
```

where network is usually *tcp*, *tcp6*,etc. But, it can be also *pipe* and *fifo*.

The *pipe* network is a network built within the application using the dial service. Addresses listening for calls are registered by calling to *NewPipe* to supply a function or connection *maker* that returns a *Conn* for a new client every time the function is called. Thus, one process might register a *foo* service in the *pipe* network and later other processes might dial *pipe!*!foo* to get a connection to the server process.

In the same way, the *fifo* network is not using the ‘‘network’’. Instead, it is a network built out of *fifos* created in the underlying system.

As an example, this dials an echo server in the *fifo* network and exercises it:

```
cc, err := Dial("fifo!*!ftest", nil)
if err != nil {
        t.Fatal(err)
}
for i := 0; i < 10; i++ {
        cc.Out <- []byte(fmt.Sprintf("<%d>", i))
        msg := string(<-cc.In)
        printf("got %s back0, msg)
}
close(cc.In)
close(cc.Out)
```

The fifo network

This network is used to let processes serve other ones that are part of different domains (different native processes on hosted environments).

The interface for the *fifo* network is exactly like the one shown for the (tcp/ip or conventional) network in a previous section. The difference is that the implementation does not rely on sockets or other network artifacts.

Instead, a server posts a *fifo* at a conventional directory and reads from it to await client calls. A client calls by creating another pair of *fifos* for input and output and writing their name to the one where the server is listening. At that point the server receives a channel-based connection that is connected to the fifos used by the client.

For example, the echo server code shown before in this paper would operate on the *fifo* network is the calls made refer to the *net/fifo* package instead of referring to the *net/svc* package. The actual implementation of the service is fully decoupled of the network used.

The client is also decoupled because it would rely on the dial service to reach the server. Needless to say that a server might listen to calls on all of the pipe, fifo, and external networks. The dial service might pick the right network to use depending on where the client and the server might be.

Multiplexed requests

Servers and clients relying on protocols that permit multiple outstanding streams of requests and replies simply call

```
mux := nchan.NewMux(con, isserver)
```

on the connection before actually using it, and then use the multiplexor to issue and receive calls or replies

```
reqc := mux.Out()
reqc.Out <- []byte("one request")
close(reqc)
```

For example, this is a full test that creates a server exporting a file tree and then issues a series of stat requests to test the server. Now how it would be very easy to stream all requests, and/or the replies. Note also how errors are propagated and checked out nicely and hwo the server and client operate on channels to talk to each other.

```
func TestSrvStat(t *testing.T) {
        os.RemoveAll(tdir)
        setup(t, tdir)
        defer os.RemoveAll(tdir)

        c1, c2 := nchan.NewConnPipe(0)

        fs, err := zx.NewLFS(tdir)
        if err != nil {
                t.Fatal(err)
        }
        srv := Serve("srv", fs, c1)
        srv.Debug = testing.Verbose()
        rfs := New(c2)
        rfs.Tag = "cli"
        rfs.Debug = testing.Verbose()

        for i := 0; i < len(stats); i++ {
                dc := rfs.Stat(stats[i].path)
                rs := ""
                if d, ok := <-dc; ok {
                        rs = d.String()
                        printf("cli: stat <%s>\n", rs)
                }
                if rs != stats[i].outs {
                        t.Fatal("wrong reply")
                }
                sts := fmt.Sprintf("%v", cerror(dc))
                printf("cli: stat sts: %s\n", sts)
                if sts != stats[i].errs {
                        t.Fatal("wrong status")
                }
        }

        srv.Close(errors.New("done"))
        rfs.Close(errors.New("done"))

        printf("all closed0)
}
```

All the pieces of this example code have been described either in this paper or in [1] or [2]. The result is simple yet effective.

As a further example, this is the main loop of the file server being tested:

```
func (s *Srv) loop() {
	c := s.c
	for x := range c.In {
		c.Debug = s.Debug
		// one req
		go func() {
			hdr, ok := <-x.In
			if !ok {
				close(x.Out, cerror(c.In))
				return
			}
			msg, err := UnpackMsg(hdr)
			if err != nil {
				close(x.Out, err)
				close(c.In, err)
				return
			}
			if op, ok := s.ops[msg.Op]; ok {
				s.dprintf("%s<- %s\n", s.Tag, msg)
				op(msg, x.In, x.Out)
				return
			}
			close(x.Out, "unknown request")
		}()
	}
	c.Close(nil)
}
```

An this is the implementation of the call tested above.

```
func (s *Srv) stat(m *Msg, c <-chan []byte, rc chan<- []byte) {
	d, err := s.t.Stat(m.Rid)
	if err != nil {
		s.dprintf("%s-> %s\n", s.Tag, err)
		close(rc, err)
		return
	}
	ds := fmt.Sprintf("%s", d)
	s.dprintf("%s-> %s\n", s.Tag, ds)
	rc <- []byte(ds)
	close(rc, nil)
}
```

The interesting point here is how the channels are used and how they fit in the system structure, not how the particular *Srv* being implemented uses its own protocol or data structures.

The implementation of the *put* operation that accepts a stream of data is as simple, yet permits streaming file contents to udpate the file. It relies on the ZX file system tools described in [2].

```
func (s *Srv) put(m *Msg, c <-chan []byte, rc chan<- []byte) {
	wt, ok := s.t.(zx.RWTree)
	if !ok {
		close(rc, "not a rw tree")
		return
	}
	err := wt.Put(m.Rid, m.D, c)
	close(rc, err)
}
```

References

1. Pipes, connections, channels and multiplexors. Francisco J. Ballesteros. Lsub TR/14/1.
2. Clive’s ZX file systems and name spaces. Francisco J. Ballesteros. Lsub TR/14/2.