Pipes, connections, channels and multiplexors

Francisco J. Ballesteros

ABSTRACT

Channels in the style of CSP are a powerful abstraction. The are close to pipes and connections used to interconnect system and network components to build distributed systems, but they are not quite the same thing. In this paper we describe changes made to Go channels and tools built upon those to provide system and network-wide services for a new OS under construction

Channels and pipes

A channel is an artifact that can be used to send (typed) data through it. The Go language operations on channels include sending, receiving, and selection among a set of send and receive operations. Go permits also to close a channel after the last item has been sent.

On most systems, processes and applications talk through pipes, network connections, FIFOS, and related artifacts. In short, they are just file descriptors once open, permit the application to write data (for sending) and/or read data (for receiving). Some of these are duplex, but they can be considered to be a pair of devices (one for each direction). In what follows we will refer to all these artifacts as pipes (e.g., a network connection may be considered as a pair of simplex pipes).

There is a mismatch between channels and pipes and this paper shows what we did to try to bridge the gap between both abstractions for a new system. The aim is to let applications leverage the CSP style of programming while, at the same time, let them work across the network.

We assume that the reader is familiar with channels in the Go language, and describes only our modifications and additions.

Close and errors

When using pipes, each end of the pipe may close and the pipe implementation takes care of propagating the error to the other end. That is not the case with standard Go channels. Furthermore, upon errors, it is desirable for one end of the pipe to learn about the error that did happen at the other end.

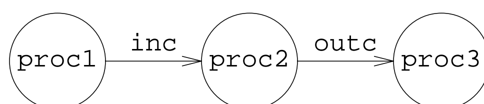
We have modified the standard Go implementation to:

1. Accept an optional *error* argument to *close*.
2. Make the send operation return *false* when used on a closed channel (instead of panicing; the receive operation already behaves nicely the case of a closed channel).
3. Provide a primitive, *cerror*, that returns the error given when the channel was closed.
4. Make a close of an already closed channel a no-operation (instead of a panic).

With this modified tool in hand, it is feasible to write the following code:

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

Here, a process consumes data from *inc* and produces new data through *outc* for another one. The image to have in mind is



where the code shown corresponds to the middle process. Perhaps the first process terminates normally (or abnormally), in which case it would close *inc*. In this case, our code closes *outc* as expected. But this time, the error given by the first process is known to the second process, and it can even forward such error to the third one.

A more interesting case is when the third process decides to cease consuming data from *outc* and calls *close*. Now, our middle process will notice that *ok* becomes *false* when it tries to send more data, and can break its loop cleanly, closing also the input channel to singal to the first process that there is no point in producing further data. In this second example, the last call to *close* is a no-operation because the output channel was already closed, and we don't need to add unnecessary code to prevent the call.

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

Channels and pipes

There are three big differences between channels and pipes (we are using *pipe* to refer to any ''file descriptor'' used to convey data, as stated before). One is that pipes may have errors when sending or receiving, but channels do not. Another one is that pipes carry ony streams of bytes and not separate messages. Yet another is that channels convey a data type but pipes convey just bytes.

The first difference is mostly dealt with the changes made to channels as described in the previous section. That is, channels may have errors while sending and or receiving, considered the changes made. Therefore, the code using a channel must consider errors in very much the same way it would do if using a pipe.

To address the third difference we are going to consider channels of *byte arrays* by now.

The second difference can be dealt with by ensuring that applications using channels to speak through a pipe preserve message boundaries within the pipe. With this in mind, a new *nchan* package provides new channel tools to bridge the gap between the channel and the pipe domains.

The following function writes each message received from *c* into *w* as it arrives. If *w* preserves message boundaries, that is enough. The second function is its conterpart.

```
func WriteBytesTo(w io.Writer, c <-chan []byte) (int64, error)
func ReadBytesFrom(r io.Reader, c chan<- []byte) (int64, error)
```

However, is most cases, the transport does not preserve message boundaries. Thus, the next function writes all messages received from *c* into *w*, but precedes each such write with a header indicating the message length. The second function can rely on this to read one message at a time and forward it to the given channel.

```
func WriteMsgsTo(w io.Writer, c <-chan []byte) (int64, error)
func ReadMsgsFrom(r io.Reader, c chan<- []byte) (int64, error)
```

One interesting feature of *WriteMsgsTo* and *ReadMsgsFrom* is that when the channel is closed, its error status is checked out and forwarded through the pipe. The other end notices that the message is an error indication and closes the channel with said error.

Thus, code like the excerpt shown for our middle process in the stream of the processes would work correctly even if the input channel comes from a pipe and not from a another process within the same program.

Connections

The *nchan* package defines a connection as

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

This joins two channels to make a full-duplex connection. A process talking to an external entity relies on this structure to bridge the system pipe used to a pair of channels. There are utilies that leverage the functions described in the previous section and build a channel interface to external pipes, for example:

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

The function creates processes to feed and drain the connection channels from and to the external pipe. Furthermore, if *rw* supports closing only for reading or writing, a close on the input or output channels would close the respective halves of the pipe. Because of the message protocol explained in the previous section, errors are also propagated across the external pipe and the process using the connection can very much ignore that the source/sink of data is external.

It is easy to build pipes where the *Out* channel sends elements through the *In* channel:

```
func NewPipe(nbuf int) Conn
```

And, using this, we can create in-memory connections that do not leave the process space:

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

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

Multiplexors

Upon the channel based connections shown in the previous sections, the *nchan* package provides multiplexors.

```
type Mux struct {
        In chan Conn
        ...
}
func NewMux(c Conn, iscaller bool) *Mux
func (m *Mux) Close(err error)
func (m *Mux) Out() chan<- []byte
func (m *Mux) Rpc() (outc chan<- []byte, repc <-chan []byte)
```

A program speaking a protocol usually creates a new *Conn* connection by dialing or accepting connections and then creates a *Mux* by calling *NewMux* to multiplex the connection among multiple requests.

The nice thing of the multiplexed connection is that requests may carry a series of messages (and not just one message per request) and may or not have replies. Replies may also be a full series of messages. Both ends of a multiplexed connetion (the process using the *mux* and its peer at the other end of the pipe) may issue requests. Thus, this is not a client-server interaction model, although it may be used as such.

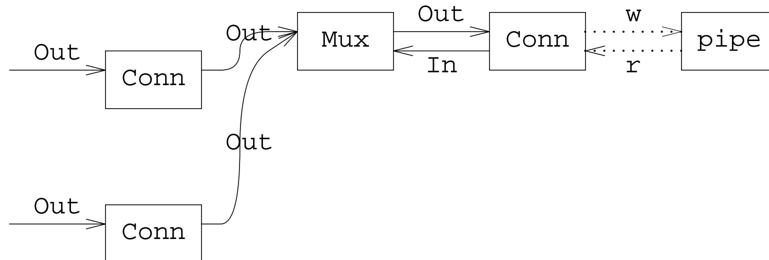
To issue new outgoing requests through the multiplexor, the process calls *Out* (to issue requests with no expected reply):

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

Or the process may call *Rpc* (to issue requests with an expected reply).

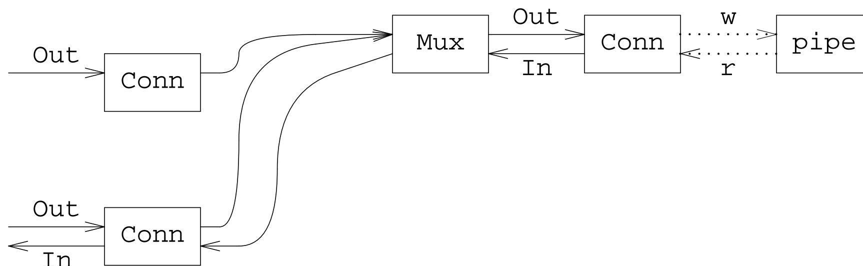
```
rc, rr := mux.Rpc()
rc <- []byte("another")
rc <- []byte("request")
close(rc)
for m := range rr {
        Printf("got %v as part of the reply\n", m)
}
Printf("and the final error status is %v\n", cerror(rr))
```

In the first case, the multiplexor returns a *Conn* to the caller with just the *Out* channel. Of course, this can be done multiple times to issue several concurrent outgoing requests:



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

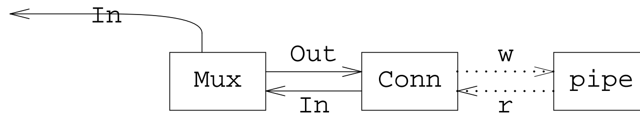
If the request depicted below requires a reply, *mux.Rpc()* is be called finstead of *mux.Out()* and the resulting picture is as shown.



The important part is that messages (and replies) sent as part of a request (or reply) may be streamed without affecting other requests and replies, other than by the usage of the underlying connection. That is, an

idle stream does not block other streams.

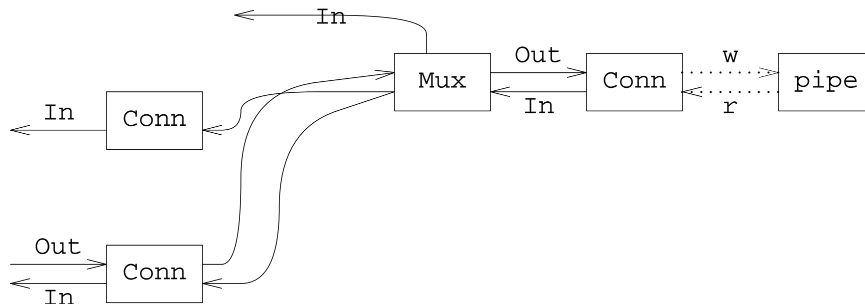
The interface for the receiving part of the multiplexor is a single *In* channel that conveys one *Conn* per incoming request. The request has only the *In* channel if no reply is expected, and has both the *In* and *Out* channels set if a reply is expected.



To receive requests from the other end of the pipe, the code might look like this:

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

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



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

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

Acknowledgements

We are very grateful to Roger Peppe and Charles Forsyth for their insights and help.