Channels done right

Francisco J. Ballesteros

ABSTRACT

Channels in the style of CSP are a powerful abstraction. The Go language includes them, inheriting much from earlier languages from Plan 9 and Research UNIX. But, to use them as a system abstraction it is necessary to be able to send in-band errors along with data and to stop senders when receivers are no longer interested in the data being sent. This paper describes Belts, a new channel abstraction that includes such features.

Channels

The Go language includes channels as a builtin type. A channel is includes in its type the data type for elements sent through it. For example,

```
var c chan int
```

declares a channel to send *int* values. They are created using *make* in either unbuffered or buffered variants. For example, this declares and creates a couple of channels:

```
unbufc := make(chan int)
bufc := make(chan int, 10)
```

Receiving from a channel blocks until an element can be received. Sending on an unbuffered channel blocks until another process receives from it. Sending on a buffered channel blocks only when the buffer is full. One operator is used to send or receive, depending on which side of the channel it is written. For example:

```
bufc <- 0       // send 0, does not block
x := <-bufc     // receive 0, copied to x.
unbufc <- 0     // send 0, blocks (no proc receiving)
```

Here the first two sentences proceed without blocking, because the message is buffered in the channel. The last blocks because nobody is receiving in this example.

There is a construct to select one among multiple send or receive operations. When no operation may proceed, the construct blocks. One some may proceed, one is executed at random and the construct terminates. For example:

```
select {
case c1 <- 3:
        // send 3 as soon as we can send to c1
case c2 <- 5:
        // send 5 as soon as we can send to c2
case x := c3:
        // receive x from c3 as soon as we can.
}
```

This constructs admits a default label to execute when no send or receive operation may be used, which leads to non-blocking variants of send and receive. In this example,

```
x := 0
select {
case x = <-c:
        // receive x from c
default:
        // didn't receive, and didn't block
}
```

we pretend that we received a zero if we couldn't receive anything.

Another feature of channels is that we may close a channel, to signal the receiver that no further data will be sent on it. In this example we send two values and close the channel, and later (or concurrently in another process) we receive from the channel until we note it is closed:

```
// send something and close it.
c <- 1
c <- 2
close(c)

// receive from c until closed or a zero value is sent.
for <-c != 0 {
}
```

Once closed, a receive returns a zero value without blocking. But we can also check if the channel was closed or if a zero value was just sent:

```
if x, ok := <-c; !ok {
        // c was closed
}
```

In this case, *ok* is set to *false* if we couldn't receive becase the channel was closed. Usually, to loop receiving the *range* operator is used instead:

```
for x := range c {
        // use x as received from c
}
```

It is aware of *close* and behaves nicely when the sender is done.

There are more features documented in the Go Programming Language Specification, but what has been said is enough to understand our motivation and the discussion that follows.

Going problems

There are problems with the behaviour of channels as described in the previous section.

One problem is that, to use channels as the primary structure used to glue different processes in one application, we should be able to stop the sender when the receiver is no longer interested in data being sent.

It has been argued that a second channel can be used to convey a termination note from the receiver of data to the producer. However, this complicates the interfaces between the elements involved. If one process is a data producer and another is a consumer, we should be able to connect them in very much the same way we do with pipes:

```
producer | consumer
```

or in this case

```
c := make(chan data)
// start a producer
go produce(c)
// start a consumer
go consume(c)
```

The code for the producer and the consumer should be as simple as follows:

```
func produce(c chan data) {
        for {
                x := make a new item
                if c <- x failed {
                        break
                }
        }
}
func consume(c chan data) {
        for x := range c {
                // use x
                if don't want to use more {
                        tell c we are done
                        break
                }
        }
}
```

The argument against notifying the sender about the stop of the receiver is that a channel should convely data only from the sender to the receiver. But, to cleanly terminate the sender if the receiver decides to stop we have to complicate the code and do one of three things:

1 Use a second channel to notify the sender that the receiver is done.

2 Spawn a new process (or use the receiver process) to consume the rest of the stream of data without actually processing it.

3 Let the sender block forever and forget about it.

The last two cases are a waste of resources, and thus should not be used in practice. In the first case, the code would be more complex:

```
c := make(chan data)
endc := make(chan bool)
// start a producer
go produce(c, endc)
// start a consumer
go consume(c, endc)
```

where

```
func produce(c chan data, endc chan bool) {
        for {
                x := make a new item
                select {
                case c <- x:
                case <-endc:
                        return
                }
        }
}
```

and

```
func consume(c chan data, endc chan bool) {
        for {
                x, ok := <-c:
                if !ok {
                        break
                }
                // use x
                if don't want more {
                        close(endc)
                        break
                }
        }
}
```

Considering this code, the connection between then sender and the receiver is now two-ways. As it would be if we could use the channel to indicate that we are done receiving, so the argument against a backward flow of information does not seem sound at this point. We are still sending information backward, but, the code is more complex and what would be a single abstraction for message passing is now two separate structures.

But there are more problems. Another important one is that if the sender fails to produce an item, the error indication is lost and can't be send to the receiver. The receiver should have a way to know that the sender had a problem, perhaps to propagate the error to others interested in the result.

To achieve this, we must further complicate the scheme to use data structure that packs either data or an error indication, and then complicate the receiver code to unpack it. Or we must use a separate error channel to convey error messages, which is even more complex.

If should be easy to let the producer notify an error to the consumer and terminate, and then let the consumer notice at each reception if it was an error or a regular data message.

Belts mark I

To fix this issues, a new abstraction, *belt* channels has been built. By now, we have not modified the language to include it, but written a package providing a data type for the new abstraction. A *belt* might be defined as

```
// An typed belt channel.
type Chan struct {
        Donec chan bool        // notify the sender that the receiver is done.
        Datac chan interface{} // send data or errors.
        /* other unexported fields */
}
```

The *Datac* conveys data and the *Donec* conveys receiver-close indications to senders. We left these fields public to let clients use *belts* in *select* constructs using more than one channel or *belt* but in the future it is likely that all this will be hidden.

A *belt* can be used to send data or errors:

```
func (b *Chan) Snd(d interface{}) error
```

Here, *d* would be the desired data type to be sent or an *error* indication. Unlike with channels, if the receiver is done, the send operation returns an error to the caller, and it decides what to do next. Perhaps stop.

The receive operation tells errors apart from data:

```
func (b *Chan) Rcv() (interface{}, error)
```

If the sender is done, an error indication is returned. In the same way, if the sender posts an error through the *belt* the receiver will get an error indication instead of data. Otherwise, a piece of data sent is received.

The following operations can be used to close a belt for sending or for receiving:

```
func (b *Chan) CloseSnd()
func (b *Chan) CloseRcv()
```

The constructor functions creates either a buffered or an unbuffered *belt* as it could be expected.

```
func New() *Chan
func NewBuffered(nbuf int) *Chan
```

Another interesting feature enabled is that belts are polymorphic, unlike channels. Any type can be sent. The language and its type assertions and reflection can be use to keep type safety, but multiple different data items can be sent easily.

A protocol can be defined in a belt so that only certain message types (plus error indications) are accepted (and other messages are rejected with error when trying to be sent). To define a protocol, the next operation accepts an array (a slice actually) of example message values, each one being a pointer to a message instance.

```
func (b *Chan) SetProto(msgptrs ...interface{})
```

In this case, the reflection interface in the language is used to accept or reject messages to be sent through the belt. This makes it easy to define protocols made by different message types without having to define facades for the set of messages.

There are wrappers that adapt belts to traditional reader and writer interfaces, so it would be easy to write to a belt, read from it, or write a belt to a writer (to copy the data streamed to an external writer).

Further wrappers know how to pipe a belt to an external connection (a writer) and how to pipe a belt from an external connection (a reader). In this case, the connection carries messages encoded as Gobs that carry any of the types defined in the belt protocol or an error indication. This permits one belt to be conveyed through a network connection or through a system file or pipe.

It would be hard to do this with standard channels, because of the problems mentioned. However, it is easy to maintain reasonable semantics that work fine in practice when writing clients and servers that use belts to and from the network. Each network connection requires two different belts if it is to be duplex.

Example

As an example, this is how our example producer and consumer processes might be written:

```
func produce(c *belt.Chan) {
        for {
                // make a new x
                if err := c.Snd(x); err != nil {
                        // couldn't send. done.
                        break
                }
                // or to send an error...
                c.Send(err)
        }
        c.CloseSnd()
}
```

and

```
func consume(c *belt.Chan) {
        for {
                x, err := c.Rcv()
                if err != nil {
                        // couldn't receive. done.
                        break
                }
                // use x
                if don't want more {
                        c.CloseRcv()
                }
        }
}
```

The result is quite similar to that of a UNIX Pipe, although belts do not kill the sender process when the receiver is gone (because they might have to terminate cleanly or might decide to do other things).

Mark I evaluation

The next is the output from the package benchmarks, which try to send 512 byte slices by different mechanisms, using go processes.

```
BenchmarkChan   20000000               134 ns/op
BenchmarkAlt     5000000               345 ns/op
BenchmarkTSend   5000000               418 ns/op
BenchmarkTWrite  5000000               723 ns/op
BenchmarkTRead   1000000              1394 ns/op
BenchmarkPipe    1000000              2476 ns/op
```

The first one uses a native channel without being able to stop the sender. That is the fastest. The second uses a *select* construct to let the receiver stop the sender. The third one uses a belt, and is not too slow when compared to the second one. We have still to optimise the code, but it is fast enough to be used in practice as it stands, compared to using a *select* directly. The extra time taken is probably due to the use of reflection.

The last three ones report the performance when using adaptors to write on the belt, or to read from the belt, and the performance of a standard Pipe as implemented by go as a reference for this case.

Belts mark II

Thanks to a discussion with Roger Peppe, belts were optimised to be defined as:

```
// An typed belt channel.
type Chan struct {
        Datac chan interface{} // send data or errors.
        /* other unexported fields */
}
```

Instead of using a separate channel to notify receiver-closes to the sender, the data channel is simply closed in that case. This triggers a panic in the sender which can be rescued by code like:

```
func dontPanic(err *error) {
        if recover() != nil {
                *err = ErrClosed
        }
}

func (b *Chan) Snd(d interface{}) (rerr error) {
        /* ... */
        defer dontPanic(&rerr)
        b.Datac <- d
        return nil
}
```

As an extra simplification, *CloseSnd* and *CloseRcv* are now gone and there is a single *Close* operation for the belt.

With this change, the benchmark output becomes as shown next. Do not compare times with those of the previous evaluation, because other things changed. What is interesting is to compare the relative performance of the different measures in this benchmark.

```
BenchmarkSend   10000000               246 ns/op
BenchmarkWrite   5000000               601 ns/op
BenchmarkRead    2000000               827 ns/op
BenchmarkPipe    1000000              1556 ns/op
BenchmarkChan   20000000               139 ns/op
BenchmarkAlt     5000000               371 ns/op
BenchmarkTSend  10000000               293 ns/op
BenchmarkTWrite  5000000               652 ns/op
BenchmarkTRead   2000000               892 ns/op
```

Acknowledgements

We are very grateful to Roger Peppe for his insights and help.

Future work

We will optimise and fine tune the interfaces for the belts in the near future, and will probably experiment by modifying the language to make belts first-class citizens, so they can be used like channels in *select* and other constructs. We will also experiment with belts used for network communication and conduct further evaluation for them.