Network channels.

Introduction

There are many models for sharing information between communicating processes. One of the more elegant is Hoare's concept of *channels*. In this, there is no shared memory, so that none of the issues of accessing common memory arise. Instead, one process will send a message along a channel to another process. Channels may be synchronous, or asynchronous, buffered or unbuffered.

Go has channels as first order data types in the language. The canonical example of using channels is Erastophene's prime sieve: one goroutine generates integers from 2 upwards. These are pumped into a series of channels that act as sieves. Each filter is distinguished by a different prime, and it removes from its stream each number that is divisible by its prime. So the '2' goroutine filters out even numbers, while the '3' goroutine filters out multiples of 3. The first number that comes out of the current set of filters must be a new prime, and this is used to start a new filter with a new channel.

The efficacy of many thousands of goroutines communicating by many thousands of channels depends on how well the implementation of these primitives is done. Go is designed to optimise these, so this type of program is feasible.

Go also supports distributed channels using the `netchan` package. But network communications are thousands of times slower than channel communications on a single computer. Running a sieve on a network over TCP would be ludicrously slow. Nevertheless, it gives a programming option that may be useful in many situations.

Go's network channel model is somewhat similar in concept to the RPC model: a server creates channels and registers them with the network channel API. A client does a lookup for channels on a server. At this point both sides have a shared channel over which they can communicate. Note that communication is one-way: if you want to send information both ways, open two channels one for each direction.

Channel server

In order to make a channel visible to clients, you need to *export* it. This is done by creating an exporter using `NewExporter` with no parameters. The server then calls `ListenAndServe` to lsiten and handle responses. This takes two parameters, the first being the underlying transport mechanism such as "tcp" and the second being the network listening address (usually just a port number.

For each channel, the server creates a normal local channel and then calls `Export` to bind this to the network channel. At the time of export, the direction of communication must be specified. Clients search for channels by name, which is a string. This is specified to the exporter.

The server then uses the local channels in the normal way, reading or writing on them. We illustrate with an "echo" server which reads lines and sends them back. It needs two channels for this. The channel that the client writes to we name "echo-out". On the server side this is a read channel. Similarly, the channel that the client reads from we call "echo-in", which is a write channel to the server.

The server program is

```
/* EchoServer
 */
package main

import (
        "fmt"
        "os"
        "old/netchan"
)

func main() {

        // exporter, err := netchan.NewExporter("tcp", ":2345")
        exporter := netchan.NewExporter()
        err := exporter.ListenAndServe("tcp", ":2345")
        checkError(err)

        echoIn := make(chan string)
        echoOut := make(chan string)
        exporter.Export("echo-in", echoIn, netchan.Send)
        exporter.Export("echo-out", echoOut, netchan.Recv)
        for {
                fmt.Println("Getting from echoOut")
                s, ok := <-echoOut
                if !ok {
                        fmt.Printf("Read from channel failed")
                        os.Exit(1)
                }
                fmt.Println("received", s)

                fmt.Println("Sending back to echoIn")
                echoIn <- s
                fmt.Println("Sent to echoIn")
        }

}

func checkError(err error) {
```

```
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Note: at the time of writing, the server will sometimes fail with an error message "netchan export: error encoding client response". This is logged as Issue 1805

Channel client

In order to find an exported channel, the client must *import* it. This is created using `Import` which takes a protocol and a network service address of "host:port". This is then used to import a network channel by name and bind it to a local channel. Note that channel variables are *references*, so you do not need to pass their addresses to functions that change them.

The following client gets two channels to and from the echo server, and then writes and reads ten messages:

```
/* EchoClient
 */
package main

import (
        "fmt"
        "old/netchan"
        "os"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "host:port")
                os.Exit(1)
        }
        service := os.Args[1]

        importer, err := netchan.Import("tcp", service)
        checkError(err)

        fmt.Println("Got importer")
        echoIn := make(chan string)
        importer.Import("echo-in", echoIn, netchan.Recv, 1)
        fmt.Println("Imported in")

        echoOut := make(chan string)
        importer.Import("echo-out", echoOut, netchan.Send, 1)
        fmt.Println("Imported out")

        for n := 0; n < 10; n++ {
```

```
                echoOut <- "hello "
                s, ok := <-echoIn
                if !ok {
                        fmt.Println("Read failure")
                        break
                }
                fmt.Println(s, n)
        }
        close(echoOut)
        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Handling Timeouts

Because these channels use the network, there is alwasy the possibility of network errors leading to timeouts. Andrew Gerrand points out a solution using timeouts: "[Set up a timeout channel.] We can then use a select statement to receive from either ch or timeout. If nothing arrives on ch after one second, the timeout case is selected and the attempt to read from ch is abandoned."

```
timeout := make(chan bool, 1)
go func() {
    time.Sleep(1e9) // one second
    timeout <- true
}()

select {
case <- ch:
    // a read from ch has occurred
case <- timeout:
    // the read from ch has timed out
}
```

Channels of channels

The online Go tutorial at http://golang.org/doc/go_tutorial.html has an example of multiplexing, where channels of channels are used. The idea is that instread of sharing one channel, a new communicator is given their own channel to have a privagye conversation. That is, a client is sent a channel from a server through a shared channel, and uses that private channel.

This doesn't work directly with network channels: a channel cannot be sent over a network channel. So we have to be a little more indirect. Each time a client connects to a server, the server builds new network channels and exports them with new names. Then it sends the *names* of these new channels to the client which imports them. It uses these new channels for communicaiton.

A server is

```
/* EchoChanServer
 */
package main

import (
        "fmt"
        "os"
        "old/netchan"
        "strconv"
)

var count int = 0

func main() {

        exporter := netchan.NewExporter()
        err := exporter.ListenAndServe("tcp", ":2345")
        checkError(err)

        echo := make(chan string)
        exporter.Export("echo", echo, netchan.Send)
        for {
                sCount := strconv.Itoa(count)
                lock := make(chan string)
                go handleSession(exporter, sCount, lock)

                <-lock
                echo <- sCount
                count++
                exporter.Drain(-1)
        }
}

func handleSession(exporter *netchan.Exporter, sCount string, lock chan
string) {
        echoIn := make(chan string)
        exporter.Export("echoIn"+sCount, echoIn, netchan.Send)

        echoOut := make(chan string)
        exporter.Export("echoOut"+sCount, echoOut, netchan.Recv)
        fmt.Println("made " + "echoOut" + sCount)

        lock <- "done"

        for {
```

```
                s := <-echoOut
                echoIn <- s
        }
        // should unexport net channels
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

and a client is

```
/* EchoChanClient
 */
package main

import (
        "fmt"
        "old/netchan"
        "os"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "host:port")
                os.Exit(1)
        }
        service := os.Args[1]

        importer, err := netchan.Import("tcp", service)
        checkError(err)

        fmt.Println("Got importer")
        echo := make(chan string)
        importer.Import("echo", echo, netchan.Recv, 1)
        fmt.Println("Imported in")

        count := <-echo
        fmt.Println(count)

        echoIn := make(chan string)
        importer.Import("echoIn"+count, echoIn, netchan.Recv, 1)

        echoOut := make(chan string)
        importer.Import("echoOut"+count, echoOut, netchan.Send, 1)

        for n := 1; n < 10; n++ {
                echoOut <- "hello "
                s := <-echoIn
                fmt.Println(s, n)
        }
        close(echoOut)
        os.Exit(0)
}
```

```
func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Conclusion

Network channels are a distributed analogue of local channels. They behave approximately the same, but due to limitations of the model some things have to be done a little differently.

Source: http://jan.newmarch.name/go/channel/chapter-channel.html