Socket-level Programming

This chapter looks at the basic techniques for network programming. It deals with host and service addressing, and then considers TCP and UDP. It shows how to build both servers and clients using the TCP and UDP Go APIs. It also looks at raw sockets, in case you need to implement your own protocol above IP.

Introduction

There are many kinds of networks in the world. These range from the very old such as serial links, through to wide area networks made from copper and fibre, to wireless networks of various kinds, both for computers and for telecommunications devices such as phones. These networks obviously differ at the physical link layer, but in many cases they also differed at higher layers of the OSI stack.

Over the years there has been a convergence to the "internet stack" of IP and TCP/UDP. For example, Bluetooth defines physical layers and protocol layers, but on top of that is an IP stack so that the same internet programming techniques can be employed on many Bluetooth devices. Similarly, developing 4G wireless phone technologies such as LTE (Long Term Evolution) will also use an IP stack.

While IP provides the networking layer 3 of the OSI stack, TCP and UDP deal with layer 4. These are not the final word, even in the interenet world: SCTP has come from the telecommunications to challenge both TCP and UDP, while to provide internet services in interplanetary space requires new, under development protocols such as DTN. Nevertheless, IP, TCP and UDP hold sway as principal networking technologies now and at least for a considerable time into the future. Go has full support for this style of programming

This chapter shows how to do TCP and UDP programming using Go, and how to use a raw socket for other protocols.

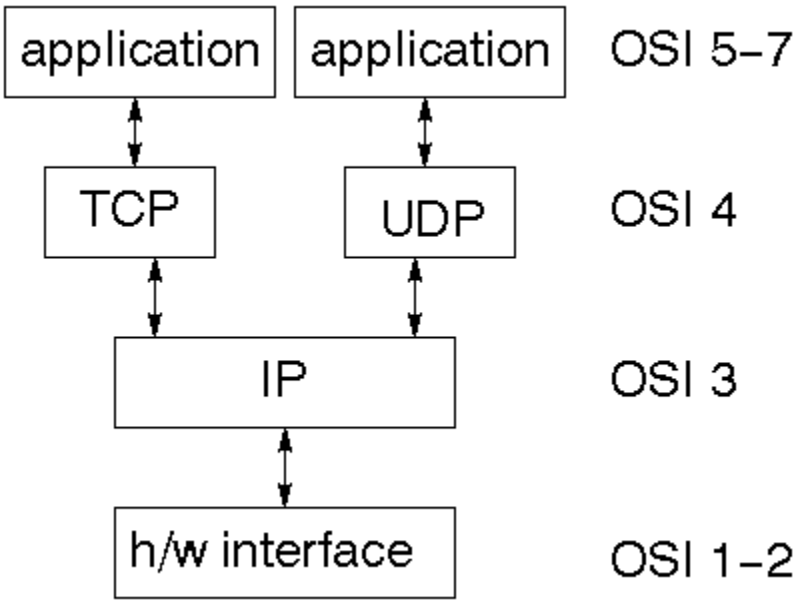
The TCP/IP stack

The OSI model was devised using a committee process wherein the standard was set up and then implemented. Some parts of the OSI standard are obscure, some parts cannot easily be implemented, some parts have not been implemented.

The TCP/IP protocol was devised through a long-running DARPA project. This worked by implementation followed by RFCs (Request For Comment). TCP/IP is the

principal Unix networking protocol. TCP/IP = Transmission Control Protocol/Internet Protocol.

The TCP/IP stack is shorter than the OSI one:



TCP is a connection-oriented protocol, UDP (User Datagram Protocol) is a connectionless protocol.

IP datagrams

The IP layer provides a connectionless and unreliable delivery system. It considers each datagram independently of the others. Any association between datagrams must be supplied by the higher layers.

The IP layer supplies a checksum that includes its own header. The header includes the source and destination addresses.

The IP layer handles routing through an Internet. It is also responsible for breaking up large datagrams into smaller ones for transmission and reassembling them at the other end.

UDP

UDP is also connectionless and unreliable. What it adds to IP is a checksum for the contents of the datagram and *port numbers*. These are used to give a client/server model - see later.

TCP

TCP supplies logic to give a reliable connection-oriented protocol above IP. It provides a *virtual circuit* that two processes can use to communicate. It also uses port numbers to identify services on a host.

Internet adddresses

In order to use a service you must be able to find it. The Internet uses an address scheme for devices such as computers so that they can be located. This addressing scheme was originally devised when there were only a handful of connected computers, and very generously allowed upto 2^32 addresses, using a 32 bit unsigned integer. These are the so-called IPv4 addresses. In recent years, the number of connected (or at least directly addressable) devices has threatened to exceed this number, and so "any day now" we will switch to IPv6 addressing which will allow upto 2^128 addresses, using an unsigned 128 bit integer. The changeover is most likely to be forced by emerging countries, as the developed world has already taken nearly all of the pool of IPv4 addresses.

IPv4 addresses

The address is a 32 bit integer which gives the IP address. This addresses down to a network interface card on a single device. The address is usually written as four bytes in decimal with a dot '.' between them, as in "127.0.0.1" or "66.102.11.104".

The IP address of any device is generally composed of two parts: the address of the network in which the device resides, and the address of the device within that network. Once upon a time, the split between network address and internal address was simple and was based upon the bytes used in the IP address.

- In a class A network, the first byte identifies the network, while the last three identify the device. There are only 128 class A networks, owned by the very early players in the internet space such as IBM, the General Electric Company and MIT (http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml)
- Class B networks use the first two bytes to identify the network and the last two to identify devices within the subnet. This allows upto 2^16 (65,536) devices on a subnet
- Class C networks use the first three bytes to identify the network and the last one to identify devices within that network. This allows upto 2^8 (actually 254, not 256) devices

This scheme doesn't work well if you want, say, 400 computers on a network. 254 is too small, while 65,536 is too large. In binary arithmetic terms, you want about 512. This can be achieved by using a 23 bit network address and 9 bits for the device addresses. Similarly, if you want upto 1024 devices, you use a 22 bit network address and a 10 bit device address.

Given an IP address of a device, and knowing how many bits N are used for the network address gives a relatively straightforward process for extracting the network address and the device address within that network. Form a "network mask" which is a 32-bit binary number with all ones in the first N places and all zeroes in the remaining ones. For example, if 16 bits are used for the network address, the mask is 11111111111111110000000000000000. It's a little inconvenient using binary, so decimal bytes are usually used. The netmask for 16 bit network addresses is 255.255.0.0, for 24 bit network addresses it is 255.255.255.0, while for 23 bit addresses it would be 255.255.254.0 and for 22 bit addresses it would be 255.255.252.0.

Then to find the network of a device, bit-wise AND it's IP address with the network mask, while the device address within the subnet is found with bit-wise AND of the 1's complement of the mask with the IP address.

IPv6 addresses

The internet has grown vastly beyond original expectations. The initially generous 32-bit addressing scheme is on the verge of running out. There are unpleasant workarounds such as NAT addressing, but eventually we will have to switch to a wider address space. IPv6 uses 128-bit addresses. Even bytes becomes cumbersome to express such addresses, so hexadecimal digits are used, grouped into 4 digits and separated by a colon ':'. A typical address might be 2002:c0e8:82e7:0:0:0:c0e8:82e7.

These addresses are not easy to remember! DNS will become even more important. There are tricks to reducing some addresses, such as eliding zeroes and repeated digits. For example, "localhost" is 0:0:0:0:0:0:0:1, which can be shortened to ::1

IP address type

The type IP

The package "net" defines many types, functions and methods of use in Go network programming. The type `IP` is defined as an array of bytes

```
      type IP []byte
```

There are several functions to manipulate a variable of type `IP`, but you are likely to use only some of them in practice. For example, the function `ParseIP(String)` will take a dotted IPv4 address or a colon IPv6 address, while the `IP` method `String` will return a string. Note that you may not get back what you started with: the string form of 0:0:0:0:0:0:0:1 is ::1.

A program to illustrate this is

```
/* IP
 */

package main

import (
        "net"
        "os"
        "fmt"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Fprintf(os.Stderr, "Usage: %s ip-addr\n", os.Args[0])
                os.Exit(1)
        }
        name := os.Args[1]

        addr := net.ParseIP(name)
        if addr == nil {
                fmt.Println("Invalid address")
        } else {
                fmt.Println("The address is ", addr.String())
        }
        os.Exit(0)
}
```

If this is compiled to the executable `IP` then it can run for example as

```
IP 127.0.0.1
```

with response

```
The address is 127.0.0.1
```

or as

```
IP 0:0:0:0:0:0:0:1
```

with response

```
The address is ::1
```

The type IPmask

In order to handle masking operations, there is the type

```
type IPMask []byte
```

There is a function to create a mask from a 4-byte IPv4 address

```
func IPv4Mask(a, b, c, d byte) IPMask
```

Alternatively, there is a method of `IP` which returns the default mask

```
func (ip IP) DefaultMask() IPMask
```

Note that the string form of a mask is a hex number such as ffff0000 for a mask of 255.255.0.0.

A mask can then be used by a method of an IP address to find the network for that IP address

```
func (ip IP) Mask(mask IPMask) IP
```

An example of the use of this is the following program:

```
/* Mask
 */

package main

import (
        "fmt"
        "net"
        "os"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Fprintf(os.Stderr, "Usage: %s dotted-ip-addr\n",
os.Args[0])
                os.Exit(1)
        }
        dotAddr := os.Args[1]
```

```
        addr := net.ParseIP(dotAddr)
        if addr == nil {
                fmt.Println("Invalid address")
                os.Exit(1)
        }
        mask := addr.DefaultMask()
        network := addr.Mask(mask)
        ones, bits := mask.Size()
        fmt.Println("Address is ", addr.String(),
                " Default mask length is ", bits,
                "Leading ones count is ", ones,
                "Mask is (hex) ", mask.String(),
                " Network is ", network.String())
        os.Exit(0)
}
```

If this is compiled to `Mask` and run by

```
Mask 127.0.0.1
```

it will return

```
Address is  127.0.0.1  Default mask length is  8  Network is  127.0.0.0
```

The type IPAddr

Many of the other functions and methods in the net package return a pointer to an `IPAddr`. This is simply a structure containing an `IP`.

```
type IPAddr {
    IP IP
}
```

A primary use of this type is to perform DNS lookups on IP host names.

```
func ResolveIPAddr(net, addr string) (*IPAddr, os.Error)
```

where `net` is one of "ip", "ip4" or "ip6". This is shown in the program

```
/* ResolveIP
 */

package main

import (
        "net"
```

```
        "os"
        "fmt"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Fprintf(os.Stderr, "Usage: %s hostname\n", os.Args[0])
                fmt.Println("Usage: ", os.Args[0], "hostname")
                os.Exit(1)
        }
        name := os.Args[1]

        addr, err := net.ResolveIPAddr("ip", name)
        if err != nil {
                fmt.Println("Resolution error", err.Error())
                os.Exit(1)
        }
        fmt.Println("Resolved address is ", addr.String())
        os.Exit(0)
}
```

Running `ResolveIP www.google.com` returns

```
Resolved address is  66.102.11.104
```

Host lookup

The function `ResolveIPAddr` will perform a DNS lookup on a hostname, and return a single IP address. However, hosts may have multiple IP addresses, usually from multiple network interface cards. They may also have multiple host names, acting as aliases.

```
func LookupHost(name string) (cname string, addrs []string, err os.Error)
```

One of these addresses will be labelled as the "canonical" host name. If you wish to find the canonical name, use `func LookupCNAME(name string) (cname string, err os.Error)`

This is shown in the following program

```
/* LookupHost
 */

package main

import (
        "net"
        "os"
```

```
        "fmt"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Fprintf(os.Stderr, "Usage: %s hostname\n", os.Args[0])
                os.Exit(1)
        }
        name := os.Args[1]

        addrs, err := net.LookupHost(name)
        if err != nil {
                fmt.Println("Error: ", err.Error())
                os.Exit(2)
        }

        for _, s := range addrs {
                fmt.Println(s)
        }
        os.Exit(0)
}
```

Note that this function returns strings, not `IPAddress` values.

Services

Services run on host machines. They are typically long lived and are designed to wait for requests and respond to them. There are many types of services, and there are many ways in which they can offer their services to clients. The internet world bases many of these services on two methods of communication, TCP and UDP, although there are other communication protocols such as SCTP waiting in the wings to take over. Many other types of service, such as peer-to-peer, remote procedure calls, communicating agents, and many others are built on top of TCP and UDP.

Ports

Services live on host machines. The IP address will locate the host. But on each computer may be many services, and a simple way is needed to distinguish between them. The method used by TCP, UDP, SCTP and others is to use a *port number*. This is an unsigned integer beween 1 and 65,535 and each service will associate itself with one or more of these port numbers.

There are many "standard" ports. Telnet usually uses port 23 with the TCP protocol. DNS uses port 53, either with TCP or with UDP. FTP uses ports 21 and 20, one for commands, the other for data transfer. HTTP usually uses port 80, but it often uses ports 8000, 8080 and 8088, all with TCP. The X Window System often takes ports 6000-6007, both on TCP and UDP.

On a Unix system, the commonly used ports are listed in the file `/etc/services`. Go has a function to interrogate this file

```
func LookupPort(network, service string) (port int, err os.Error)
```

The network argument is a string such as "tcp" or "udp", while the service is a string such as "telnet" or "domain" (for DNS).

A program using this is

```
/* LookupPort
 */

package main

import (
        "net"
        "os"
        "fmt"
)

func main() {
        if len(os.Args) != 3 {
                fmt.Fprintf(os.Stderr,
                        "Usage: %s network-type service\n",
                        os.Args[0])
                os.Exit(1)
        }
        networkType := os.Args[1]
        service := os.Args[2]

        port, err := net.LookupPort(networkType, service)
        if err != nil {
                fmt.Println("Error: ", err.Error())
                os.Exit(2)
        }

        fmt.Println("Service port ", port)
        os.Exit(0)
}
```

For example, running `LookupPort tcp telnet` prints `Service port: 23`

The type TCPAddr

The type `TCPAddr` is a structure containing an `IP` and a `port`:

```
type TCPAddr struct {
```

```
    IP   IP
    Port int
}
```

The function to create a `TCPAddr` is `ResolveTCPAddr`

```
func ResolveTCPAddr(net, addr string) (*TCPAddr, os.Error)
```

where `net` is one of "tcp", "tcp4" or "tcp6" and the `addr` is a string composed of a host name or IP address, followed by the port number after a ":", such as "www.google.com:80" or '127.0.0.1:22". if the address is an IPv6 address, which already has colons in it, then the host part must be enclosed in square brackets, such as "[::1]:23". Another special case is often used for servers, where the host address is zero, so that the TCP address is really just the port name, as in ":80" for an HTTP server.

TCP Sockets

When you know how to reach a service via its network and port IDs, what then? If you are a client you need an API that will allow you to connect to a service and then to send messages to that service and read replies back from the service.

If you are a server, you need to be able to bind to a port and listen at it. When a message comes in you need to be able to read it and write back to the client.

The `net.TCPConn` is the Go type which allows full duplex communication between the client and the server. Two major methods of interest are

```
func (c *TCPConn) Write(b []byte) (n int, err os.Error)
func (c *TCPConn) Read(b []byte) (n int, err os.Error)
```

A `TCPConn` is used by both a client and a server to read and write messages.

TCP client

Once a client has established a TCP address for a service, it "dials" the service. If succesful, the dial returns a `TCPConn` for communication. The client and the server exchange messages on this. Typically a client writes a request to the server using the `TCPConn`, and reads a response from the `TCPConn`. This continues until either (or both) sides close the connection. A TCP connection is established by the client using the function

```
func DialTCP(net string, laddr, raddr *TCPAddr) (c *TCPConn, err os.Error)
```

where `laddr` is the local address which is usually set to `nil` and `raddr` is the remote address of the service, and the `net` string is one of "tcp4", "tcp6" or "tcp" depending on whether you want a TCPv4 connection, a TCPv6 connection or don't care.

A simple example can be provided by a client to a web (HTTP) server. We will deal in substantially more detail with HTTP clients and servers in a later chapter, but for now we will keep it simple.

One of the possible messages that a client can send is the "HEAD" message. This queries a server for information about the server and a document on that server. The server returns information, but does not return the document itself. The request sent to query an HTTP server could be

```
"HEAD / HTTP/1.0\r\n\r\n"
```

which asks for information about the root document and the server. A typical response might be

```
HTTP/1.0 200 OK
ETag: "-9985996"
Last-Modified: Thu, 25 Mar 2010 17:51:10 GMT
Content-Length: 18074
Connection: close
Date: Sat, 28 Aug 2010 00:43:48 GMT
Server: lighttpd/1.4.23
```

We first give the program (GetHeadInfo.go) to establish the connection for a TCP address, send the request string, read and print the response. Once compiled it can be invoked by e.g.

```
GetHeadInfo www.google.com:80
```

The program is

```
/* GetHeadInfo
 */
package main

import (
        "net"
        "os"
        "fmt"
        "io/ioutil"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Fprintf(os.Stderr, "Usage: %s host:port ", os.Args[0])
```

```
                os.Exit(1)
        }
        service := os.Args[1]

        tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
        checkError(err)

        conn, err := net.DialTCP("tcp", nil, tcpAddr)
        checkError(err)

        _, err = conn.Write([]byte("HEAD / HTTP/1.0\r\n\r\n"))
        checkError(err)

        //result, err := readFully(conn)
        result, err := ioutil.ReadAll(conn)
        checkError(err)

        fmt.Println(string(result))

        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}
```

The first point to note is the almost excessive amount of error checking that is going on. This is normal for networking programs: the opportunities for failure are substantially greater than for standalone programs. Hardware may fail on the client, the server, or on any of the routers and switches in the middle; communication may be blocked by a firewall; timeouts may occur due to network load; the server may crash while the client is talking to it. The following checks are performed:

1. There may be syntax errors in the address specified
2. The attempt to connect to the remote service may fail. For example, the service requested might not be running, or there may be no such host connected to the network
3. Although a connection has been established, writes to the service might fail if the connection has died suddenly, or the network times out
4. Similarly, the reads might fail

Reading from the server requires a comment. In this case, we read essentially a single response from the server. This will be terminated by end-of-file on the connection. However, it may consist of several TCP packets, so we need to keep reading till the end of file. The `io/ioutil` function `ReadAll` will look after these issues and return the complete response. (Thanks to Roger Peppe on the golang-nuts mailing list.).

There are some language issues involved. First, most of the functions return a dual value, with possible error as second value. If no error occurs, then this will be `nil`. In C, the same behaviour is gained by special values such as `NULL`, or -1, or zero being returned - if that is possible. In Java, the same error checking is managed by throwing and catching exceptions, which can make the code look very messy.

In earlier versions of this program, I returned the result in the array `buf`, which is of type `[512]byte`. Attempts to coerce this to a string failed - only byte arrays of type `[]byte` can be coerced. This is a bit of a nuisance.

A Daytime server

About the simplest service that we can build is the daytime service. This is a standard Internet service, defined by RFC 867, with a default port of 13, on both TCP and UDP. Unfortunately, with the (justified) increase in paranoia over security, hardly any sites run a daytime server any more. Never mind, we can build our own. (For those interested, if you install `inetd` on your system, you usually get a daytime server thrown in.)

A server registers itself on a port, and listens on that port. Then it blocks on an "accept" operation, waiting for clients to connect. When a client connects, the accept call returns, with a connection object. The daytime service is very simple and just writes the current time to the client, closes the connection, and resumes waiting for the next client.

The relevant calls are

```
func ListenTCP(net string, laddr *TCPAddr) (l *TCPListener, err os.Error)
func (l *TCPListener) Accept() (c Conn, err os.Error)
```

The argument `net` can be set to one of the strings "tcp", "tcp4" or "tcp6". The IP address should be set to zero if you want to listen on all network interfaces, or to the IP address of a single network interface if you only want to listen on that interface. If the port is set to zero, then the O/S will choose a port for you. Otherwise you can choose your own. Note that on a Unix system, you cannot listen on a port below 1024 unless you are the system supervisor, root, and ports below 128 are standardised by the IETF. The example program chooses port 1200 for no particular reason. The TCP address is given as ":1200" - all interfaces, port 1200.

The program is

```
/* DaytimeServer
 */
```

```
package main

import (
        "fmt"
        "net"
        "os"
        "time"
)

func main() {

        service := ":1200"
        tcpAddr, err := net.ResolveTCPAddr("ip4", service)
        checkError(err)

        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)

        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }

                daytime := time.Now().String()
                conn.Write([]byte(daytime)) // don't care about return value
                conn.Close()                // we're finished with this client
        }
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}
```

If you run this server, it will just wait there, not doing much. When a client connects to it, it will respond by sending the daytime string to it and then return to waiting for the next client.

Note the changed error handling in the server as compared to a client. The server should run forever, so that if any error occurs with a client, the server just ignores that client and carries on. A client could otherwise try to mess up the connection with the server, and bring it down!

We haven't built a client. That is easy, just changing the previous client to omit the initial write. Alternatively, just open up a `telnet` connection to that host:

```
telnet localhost 1200
```

This will produce output such as

```
$telnet localhost 1200
Trying ::1...
Connected to localhost.
Escape character is '^]'.
Sun Aug 29 17:25:19 EST 2010Connection closed by foreign host.
```

where "Sun Aug 29 17:25:19 EST 2010" is the output from the server.

Multi-threaded server

"echo" is another simple IETF service. This just reads what the client types, and sends it back:

```
/* SimpleEchoServer
 */
package main

import (
        "net"
        "os"
        "fmt"
)

func main() {

        service := ":1201"
        tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
        checkError(err)

        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)

        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }
                handleClient(conn)
                conn.Close() // we're finished
        }
}

func handleClient(conn net.Conn) {
        var buf [512]byte
        for {
                n, err := conn.Read(buf[0:])
                if err != nil {
                        return
                }
                fmt.Println(string(buf[0:]))
                _, err2 := conn.Write(buf[0:n])
                if err2 != nil {
                        return
                }
```

```
        }
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}
```

While it works, there is a significant issue with this server: it is single-threaded. While a client has a connection open to it, no other cllient can connect. Other clients are blocked, and will probably time out. Fortunately this is easly fixed by making the client handler a go-routine. We have also moved the connection close into the handler, as it now belongs there

```
/* ThreadedEchoServer
 */
package main

import (
        "net"
        "os"
        "fmt"
)

func main() {

        service := ":1201"
        tcpAddr, err := net.ResolveTCPAddr("ip4", service)
        checkError(err)

        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)

        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }
                // run as a goroutine
                go handleClient(conn)
        }
}

func handleClient(conn net.Conn) {
        // close connection on exit
        defer conn.Close()

        var buf [512]byte
        for {
                // read upto 512 bytes
                n, err := conn.Read(buf[0:])
```

```
                if err != nil {
                        return
                }

                // write the n bytes read
                _, err2 := conn.Write(buf[0:n])
                if err2 != nil {
                        return
                }
        }
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}
```

Controlling TCP connections

Timeout

The server may wish to timeout a client if it does not respond quickly enough i.e. does not write a request to the server in time. This should be a long period (several minutes), because the user may be taking their time. Conversely, the client may want to timeout the server (after a much shorter time). Both do this by

```
func (c *TCPConn) SetTimeout(nsec int64) os.Error
```

before any reads or writes on the socket.

Staying alive

A client may wish to stay connected to a server even if it has nothing to send. It can use

```
func (c *TCPConn) SetKeepAlive(keepalive bool) os.Error
```

There are several other connection control methods, documented in the "net" package.

UDP Datagrams

In a connectionless protocol each message contains information about its origin and destination. There is no "session" established using a long-lived socket. UDP clients and servers make use of datagrams, which are individual messages containing source and destination information. There is no state maintained by these messages, unless

the client or server does so. The messages are not guaranteed to arrive, or may arrive out of order.

The most common situation for a client is to send a message and hope that a reply arrives. The most common situation for a server would be to receive a message and then send one or more replies back to that client. In a peer-to-peer situation, though, the server may just forward messages to other peers.

The major difference between TCP and UDP handling for Go is how to deal with packets arriving from possibly multiple clients, without the cushion of a TCP session to manage things. The major calls needed are

```
func ResolveUDPAddr(net, addr string) (*UDPAddr, os.Error)
func DialUDP(net string, laddr, raddr *UDPAddr) (c *UDPConn, err os.Error)
func ListenUDP(net string, laddr *UDPAddr) (c *UDPConn, err os.Error)
func (c *UDPConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr, err os.Error
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (n int, err os.Error)
```

The client for a UDP time service doesn't need to make many changes, just changing `...TCP...` calls to `...UDP...` calls:

```
/* UDPDaytimeClient
 */
package main

import (
        "net"
        "os"
        "fmt"
)

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

        udpAddr, err := net.ResolveUDPAddr("up4", service)
        checkError(err)

        conn, err := net.DialUDP("udp", nil, udpAddr)
        checkError(err)

        _, err = conn.Write([]byte("anything"))
        checkError(err)

        var buf [512]byte
        n, err := conn.Read(buf[0:])
```

```
        checkError(err)

        fmt.Println(string(buf[0:n]))

        os.Exit(0)
}

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

while the server has to make a few more:

```
/* UDPDaytimeServer
 */
package main

import (
        "fmt"
        "net"
        "os"
        "time"
)

func main() {

        service := ":1200"
        udpAddr, err := net.ResolveUDPAddr("up4", service)
        checkError(err)

        conn, err := net.ListenUDP("udp", udpAddr)
        checkError(err)

        for {
                handleClient(conn)
        }
}

func handleClient(conn *net.UDPConn) {

        var buf [512]byte

        _, addr, err := conn.ReadFromUDP(buf[0:])
        if err != nil {
                return
        }

        daytime := time.Now().String()

        conn.WriteToUDP([]byte(daytime), addr)
}

func checkError(err error) {
        if err != nil {
```

```
                fmt.Fprintf(os.Stderr, "Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Server listening on multiple sockets

A server may be attempting to listen to multiple clients not just on one port, but on many. In this case it has to use some sort of polling mechanism between the ports.

In C, the select() call lets the kernel do this work. The call takes a number of file descriptors. The process is suspended. When I/O is ready on one of these, a wakeup is done, and the process can continue. This is cheaper than busy polling. In Go, accomplish the same by using a different goroutine for each port. A thread will become runnable when the lower-level select() discovers that I/O is ready for this thread.

The types Conn, PacketConn and Listener

So far we have differentiated between the API for TCP and the API for UDP, using for example `DialTCP` and `DialUDP` returning a `TCPConn` and `UDPConn` respectively. The type `Conn` is an interface and both `TCPConn`and `UDPConn` implement this interface. To a large extent you can deal with this interface rather than the two types.

Instead of separate dial functions for TCP and UDP, you can use a single function

```
func Dial(net, laddr, raddr string) (c Conn, err os.Error)
```

The `net` can be any of "tcp", "tcp4" (IPv4-only), "tcp6" (IPv6-only), "udp", "udp4" (IPv4-only), "udp6" (IPv6-only), "ip", "ip4" (IPv4-only) and "ip6" IPv6-only). It will return an appropriate implementation of the `Conn`interface. Note that this function takes a string rather than address as `raddr` argument, so that programs using this can avoid working out the address type first.

Using this function makes minor changes to programs. For example, the earlier program to get HEAD information from a Web page can be re-written as

```
/* IPGetHeadInfo
 */
package main

import (
```

```
        "bytes"
        "fmt"
        "io"
        "net"
        "os"
)

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

        conn, err := net.Dial("tcp", service)
        checkError(err)

        _, err = conn.Write([]byte("HEAD / HTTP/1.0\r\n\r\n"))
        checkError(err)

        result, err := readFully(conn)
        checkError(err)

        fmt.Println(string(result))

        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}

func readFully(conn net.Conn) ([]byte, error) {
        defer conn.Close()

        result := bytes.NewBuffer(nil)
        var buf [512]byte
        for {
                n, err := conn.Read(buf[0:])
                result.Write(buf[0:n])
                if err != nil {
                        if err == io.EOF {
                                break
                        }
                        return nil, err
                }
        }
        return result.Bytes(), nil
}
```

Writing a server can be similarly simplified using the function

```
```

```
func Listen(net, laddr string) (l Listener, err os.Error)
```

which returns an object implementing the `Listener` interface. This interface has a method

```
func (l Listener) Accept() (c Conn, err os.Error)
```

which will allow a server to be built. Using this, the multi-threaded Echo server given earlier becomes

```
/* ThreadedIPEchoServer
 */
package main

import (
        "fmt"
        "net"
        "os"
)

func main() {

        service := ":1200"
        listener, err := net.Listen("tcp", service)
        checkError(err)

        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }
                go handleClient(conn)
        }
}

func handleClient(conn net.Conn) {
        defer conn.Close()

        var buf [512]byte
        for {
                n, err := conn.Read(buf[0:])
                if err != nil {
                        return
                }
                _, err2 := conn.Write(buf[0:n])
                if err2 != nil {
                        return
                }
        }
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
```

```
        }
}
```

If you want to write a UDP server, then there is an interface `PacketConn` and a method to return an implementation of this:

```
func ListenPacket(net, laddr string) (c PacketConn, err os.Error)
```

This interface has primary methods `ReadFrom` and `WriteTo` to handle packet reads and writes.

The Go `net` package recommends using these interface types rather than the concrete ones. But by using them, you lose specific methods such as `SetKeepAlive` or `TCPConn` and `SetReadBuffer` of `UDPConn`, unless you do a type cast. It is your choice.

Raw sockets and the type IPConn

This section covers advanced material which most programmers are unlikely to need. it deals with *raw sockets*, which allow the programmer to build their own IP protocols, or use protocols other than TCP or UDP

TCP and UDP are not the only protocols built above the IP layer. The site http://www.iana.org/assignments/protocol-numbers lists about 140 of them (this list is often available on Unix systems in the file `/etc/protocols`). TCP and UDP are only numbers 6 and 17 respectively on this list.

Go allows you to build so-called raw sockets, to enable you to communicate using one of these other protocols, or even to build your own. But it gives minimal support: it will connect hosts, and write and read packets between the hosts. In the next chapter we will look at designing and implementing your own protocols above TCP; this section considers the same type of problem, but at the IP layer.

To keep things simple, we shall use almost the simplest possible example: how to send a ping message to a host. Ping uses the "echo" command from the ICMP protocol. This is a byte-oriented protocol, in which the client sends a stream of bytes to another host, and the host replies. the format is:

- The first byte is 8, standing for the echo message
- The second byte is zero
- The third and fourth bytes are a checksum on the entire message
- The fifth and sixth bytes are an arbitrary indentifier
- The seventh and eight bytes are an arbitrary sequence number

- The rest of the packet is user data

The following program will prepare an IP connection, send a ping request to a host and get a reply. You may need to have root access in order to run it successfully.

```
/* Ping
 */
package main

import (
        "bytes"
        "fmt"
        "io"
        "net"
        "os"
)

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

        addr, err := net.ResolveIPAddr("ip", os.Args[1])
        if err != nil {
                fmt.Println("Resolution error", err.Error())
                os.Exit(1)
        }

        conn, err := net.DialIP("ip4:icmp", addr, addr)
        checkError(err)

        var msg [512]byte
        msg[0] = 8  // echo
        msg[1] = 0  // code 0
        msg[2] = 0  // checksum, fix later
        msg[3] = 0  // checksum, fix later
        msg[4] = 0  // identifier[0]
        msg[5] = 13 //identifier[1]
        msg[6] = 0  // sequence[0]
        msg[7] = 37 // sequence[1]
        len := 8

        check := checkSum(msg[0:len])
        msg[2] = byte(check >> 8)
        msg[3] = byte(check & 255)

        _, err = conn.Write(msg[0:len])
        checkError(err)

        _, err = conn.Read(msg[0:])
        checkError(err)

        fmt.Println("Got response")
```

```
        if msg[5] == 13 {
                fmt.Println("identifier matches")
        }
        if msg[7] == 37 {
                fmt.Println("Sequence matches")
        }

        os.Exit(0)
}

func checkSum(msg []byte) uint16 {
        sum := 0

        // assume even for now
        for n := 1; n < len(msg)-1; n += 2 {
                sum += int(msg[n])*256 + int(msg[n+1])
        }
        sum = (sum >> 16) + (sum & 0xffff)
        sum += (sum >> 16)
        var answer uint16 = uint16(^sum)
        return answer
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}

func readFully(conn net.Conn) ([]byte, error) {
        defer conn.Close()

        result := bytes.NewBuffer(nil)
        var buf [512]byte
        for {
                n, err := conn.Read(buf[0:])
                result.Write(buf[0:n])
                if err != nil {
                        if err == io.EOF {
                                break
                        }
                        return nil, err
                }
        }
        return result.Bytes(), nil
}
```

Conclusion

This chapter has considered programming at the IP, TCP and UDP levels. This is often necessary if you wish to implement your own protocol, or build a client or server for an existing protocol.

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