Network programming with Go

Jan Newmarch

Table of Contents

Network Programming with Go by Jan Newmarch

An e-book on building network applications using Google's Go programming language (Golang)

This book is hosted on github-pages.

Ebook can be downloaded as pdf, epub and mobi.

Chapter 1 Architecture

You can't build a system without some idea of what you want to build. And you can't build it if you don't know the environment in which it will work.

GUI programs are different to batch processing programs; games programs are different to business programs; and distributed programs are different to standalone programs.

They each have their approaches, their common patterns, the problems that typically arise and the solutions that are often used.

This chapter covers the high level architectural aspects of distributed systems. There are many ways of looking at such systems, and many of these are dealt with.

Protocol Layers

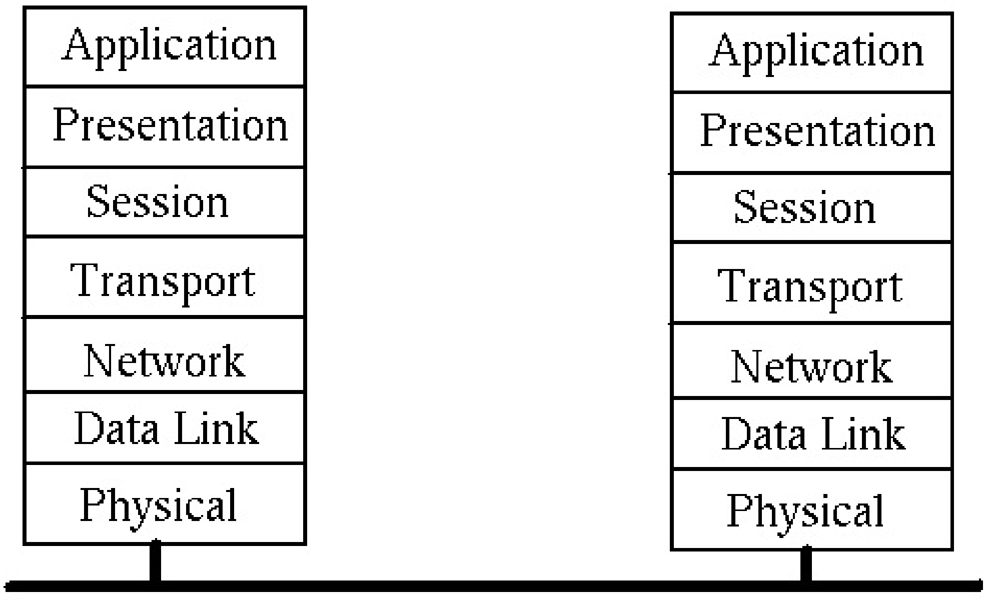
Distributed systems are hard. There are multiple computers involved, which have to be connected in some way. Programs have to be written to run on each computer in the system and they all have to co-operate to get a distributed task done.

The common way to deal with complexity is to break it down into smaller and simpler parts. These parts have their own structure, but they also have defined means of communicating with other related parts. In distributed systems, the parts are called protocol layers and they have clearly defined functions. They form a stack, with each layer communicating with the layer above and the layer below. The communication between layers is defined by protocols.

Network communications requires protocols to cover high-level application communication all the way down to wire communication and the complexity handled by encapsulation in protocol layers.

ISO OSI Protocol

Although it was never properly implemented, the OSI (Open Systems Interconnect) protocol has been a major influence in ways of talking about and influencing distributed systems design. It is commonly given in the following figure:



OSI layers

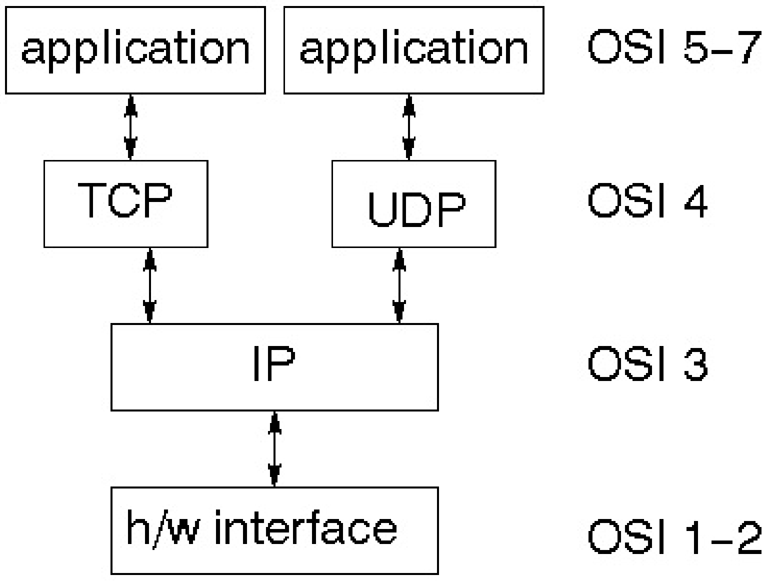
The function of each layer is:

- Network layer provides switching and routing technologies
- Transport layer provides transparent transfer of data between end systems and is responsible for end-to-end error recovery and flow control

- Session layer establishes, manages and terminates connections between applications.
- Presentation layer provides independence from differences in data representation (e.g. encryption)
- Application layer supports application and end-user processes

TCP/IP Protocol

While the OSI model was being argued, debated, partly implemented and fought over, the DARPA internet research project was busy building the TCP/IP protocols. These have been immensely successful and have led to The Internet (with capitals). This is a much simpler stack:



Some Alternative Protocols

Although it almost seems like it, the TCP/IP protocols are not the only ones in existence and in the long run may not even be the most successful. There are many protocols occupying significant niches, such as

- Firewire
- USB
- Bluetooth
- WiFi

There is active work continuing on many other protocols, even quite bizarre ones such as those for the "internet in space."

The focus in this book will be on the TCP/IP, but you should be aware of these other ones.

Networking

A network is a communications system for connecting end systems called hosts. The mechanisms of connection might be copper wire, ethernet, fiber optic or wireless, but that won't concern us here. A local area network (LAN) connects computers that are close together, typically belonging to a home, small organization or part of a larger organization.

A Wide Area Network (WAN) connects computers across a larger physical area, such as between cities. There are other types as well, such as MANs (Metropolitan Area Network), PANs (Personal Area Networks) and even BANs (Body Area Network).

An internet is a connection of two or more distinct networks, typically LANs or WANs. An intranet is an internet with all networks belonging to a single organization.

There are significant differences between an internet and an intranet. Typically an intranet will be under a single administrative control, which will impose a single set of coherent policies. An internet on the other hand will not be under the control of a single body, and the controls exercised over different parts may not even be compatible.

A trivial example of such differences is that an intranet will often be restricted to computers by a small number of vendors running a standardized version of a particular operating system. On the other hand, an internet will often have a smorgasbord of different computers and operating systems.

The techniques of this book will be applicable to internets. They will also be valid for intranets, but there you will also find specialized, non-portable systems.

And then there is the "mother" of all internets: The Internet. This is just a very, very large internet that connects us to Google, my computer to your computer and so on.

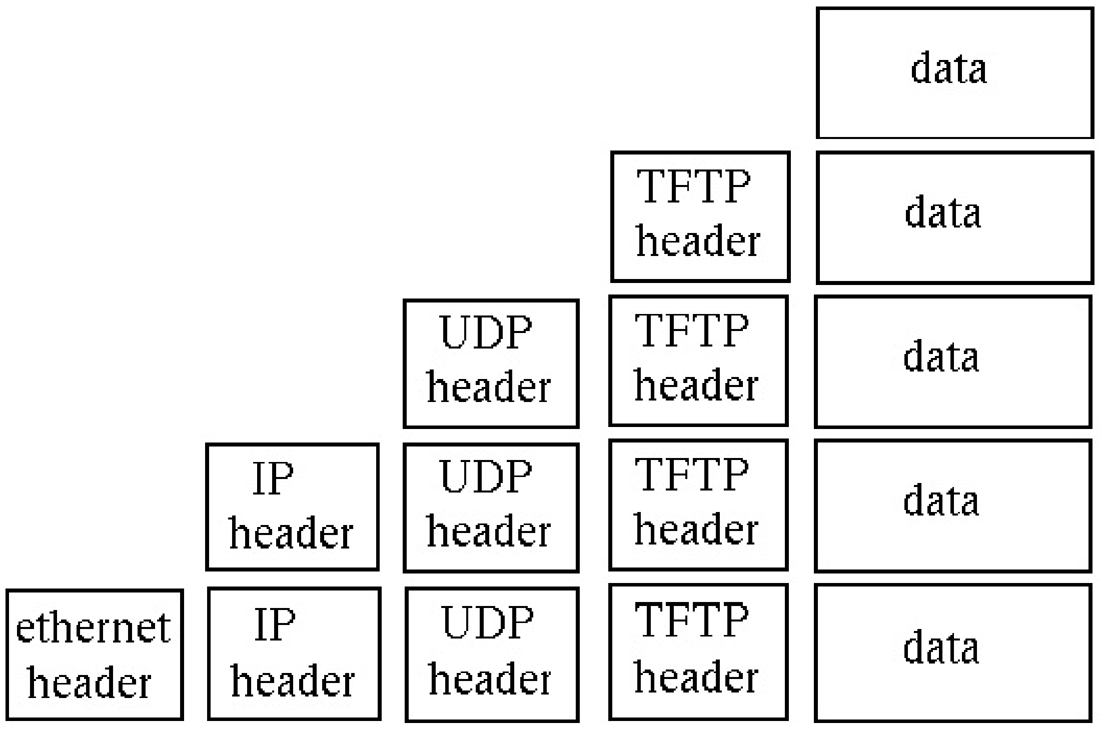
Gateways

A gateway is a generic term for an entity used to connect two or more networks. A repeater operates at the physical level and copies the information from one subnet to another. A bridge operates at the data link layer level and copies frames between networks. A router operates at the network level and not only moves information between networks but also decides on the route.

Packet encapsulation

The communication between layers in either the OSI or the TCP/IP stacks is done by sending packets of data from one layer to the next, and then eventually across the network. Each layer has administrative information that it has to keep about its own layer. It does this by adding header information to the packet it receives from the layer above, as the packet passes down. On the receiving side, these headers are removed as the packet moves up.

For example, the TFTP (Trivial File Transfer Protocol) moves files from one computer to another. It uses the UDP protocol on top of the IP protocol, which may be sent over Ethernet. This looks like:



The packet transmitted over ethernet, is of course the bottom one.

Connection Models

In order for two computers to communicate, they must set up a path whereby they can send at least one message in a session. There are two major models for this:

- Connection oriented
- Connectionless

Connection oriented

A single connection is established for the session. Two-way communications flow along the connection. When the session is over, the connection is broken. The analogy is to a phone conversation. An example is TCP

Connectionless

In a connectionless system, messages are sent independent of each other. Ordinary mail is the analogy. Connectionless messages may arrive out of order. An example is the IP protocol.

Connection oriented transports may be established on top of connectionless ones - TCP over IP. Connectionless transports may be established on top of connection oriented ones - HTTP over TCP.

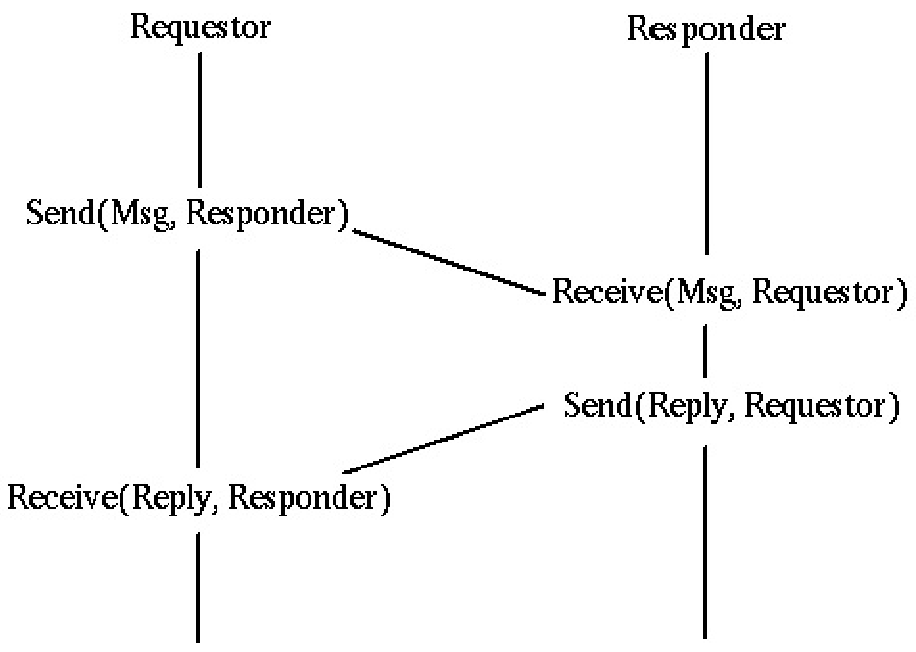
There can be variations on these. For example, a session might enforce messages arriving, but might not guarantee that they arrive in the order sent. However, these two are the most common.

Communications Models

Message passing

Some non-procedural languages are built on the principle of *message passing*. Concurrent languages often use such a mechanism, and the most well known example is probably the Unix pipeline. The Unix pipeline is a pipeline of bytes, but there is not an inherent limitation: Microsoft's PowerShell can send objects along its pipelines, and concurrent languages such as Parlog could send arbitrary logic data structures in messages between concurrent processes.

Message passing is a primitive mechanism for distributed systems. Set up a connection and pump some data down it. At the other end, figure out what the message was and respond to it, possibly sending messages back. This is illustrated by



Low level event driven systems such as the X Window System function in a somewhat similar way: wait for message from a user (mouse clicks, etc), decode them and act on them.

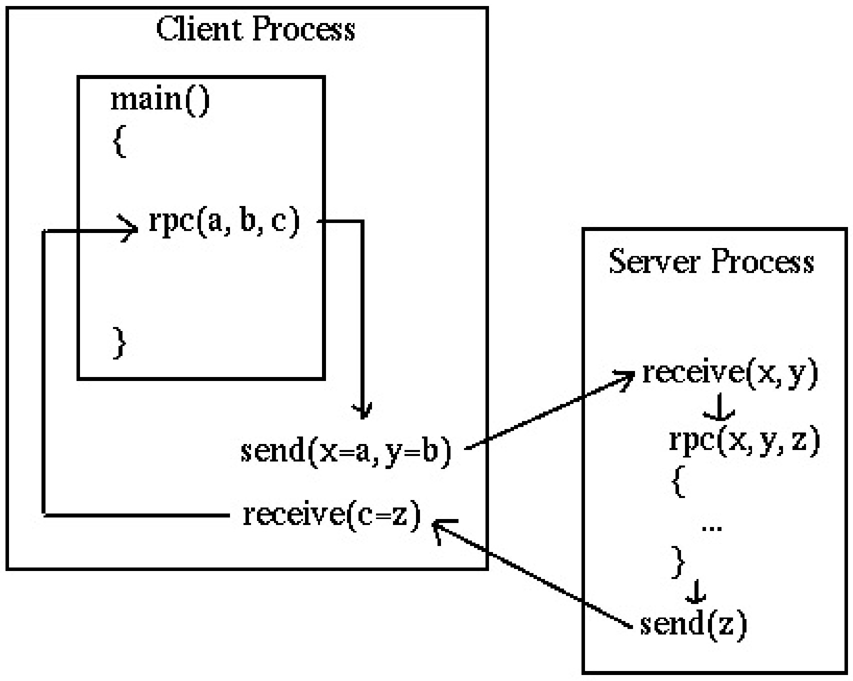
Higher level event driven systems assume that this decoding has been done by the underlying system and the event is then dispatched to an appropriate object such as a ButtonPress handler. This can also be done in distributed message passing systems, whereby a message received across the network is partly decoded and dispatched to an appropriate handler.

Remote procedure call

In any system, there is a transfer of information and flow control from one part of the system to another. In procedural languages this may consist of the procedure call, where information is placed on a call stack and then control flow is transferred to another part of the program.

Even with procedure calls, there are variations. The code may be statically linked so that control transfers from one part of the program's executable code to another part. Due to the increasing use of library routines, it has become commonplace to have such code in dynamic link libraries (DLL-s), where control transfers to an independent piece of code.

DLLs run in the same machine as the calling code. It is a simple (conceptual) step to transfer control to a procedure running in a different machine. The mechanics of this are not so simple! However, this model of control has given rise to the "remote procedure call" (RPC) which is discussed in much detail in a later chapter. This is illustrated by



There is an historical oddity called the "lightweight remote procedure call" invented by Microsoft as they transitioned from 16-bit to 32-bit applications. A 16-bit application might need to transfer data to a 32-bit application *on the same machine*. That made it lightweight as there was no networking! But it had many of the other issues of RPC systems in data representations and conversion.

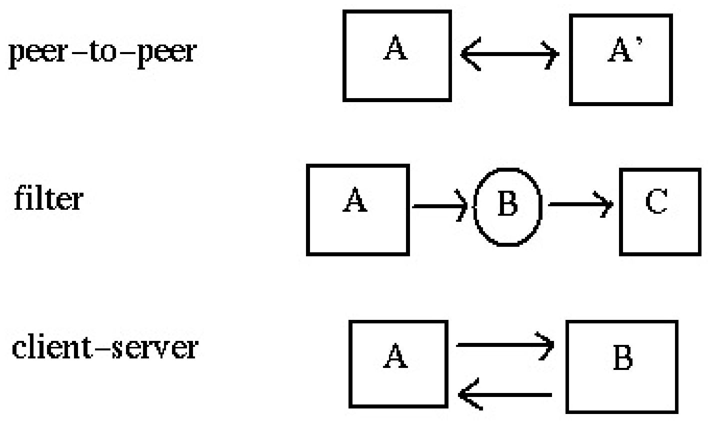
Distributed Computing Models

At the highest level, we could consider the equivalence or the non-equivalence of components of a distributed system. The most common occurrence is an asymmetric one: a client sends requests to a server, and the server responds. This is a *client-server* system.

If both components are equivalent, both able to initiate and to respond to messages, then we have a *peer-to-peer* system. Note that this is a logical classification: one peer may be a 16,000 core mainframe, the other might be a mobile phone. But if both can act similarly then they are peers.

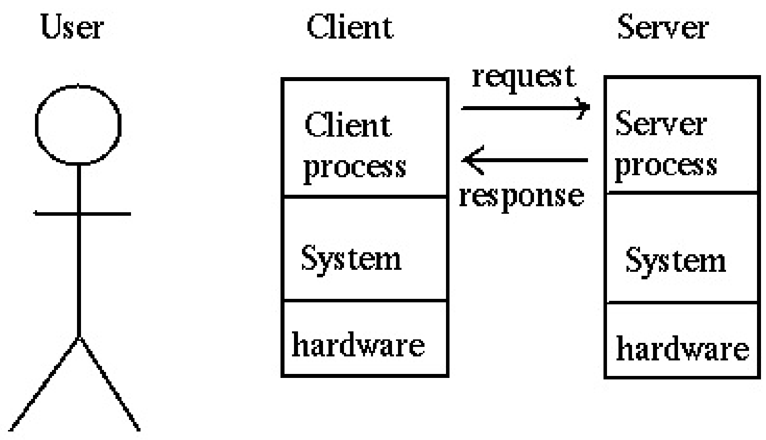
A third model is the so-called *filter*. Here one component passes information to another which modifies it before passing it to a third. This is a fairly common model: for example, the middle component gets information from a database as SQL records and transforms it into an HTML table for the third component (which might be a browser).

These are illustrated as:



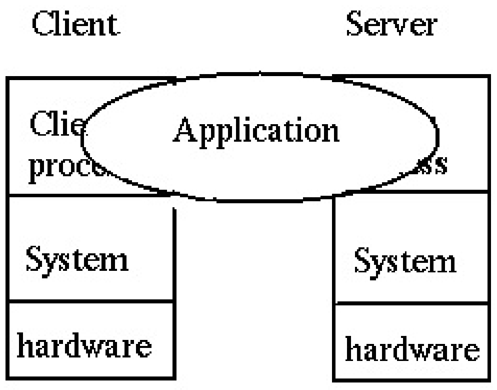
Client/Server System

Another view of a client/server system is



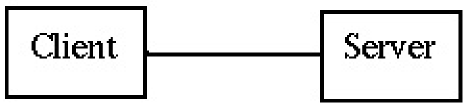
Client/Server Application

And a third view is

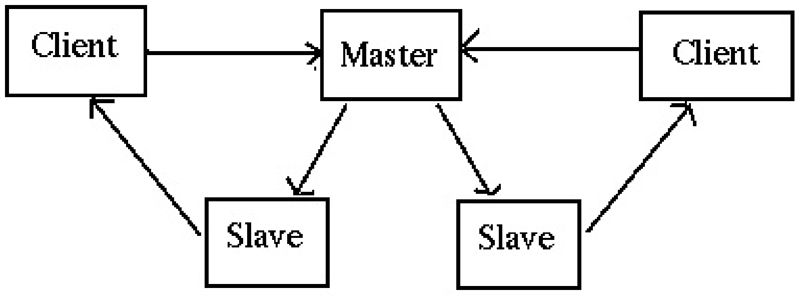


Server Distribution

A client-server systems need not be simple. The basic model is single client, single server

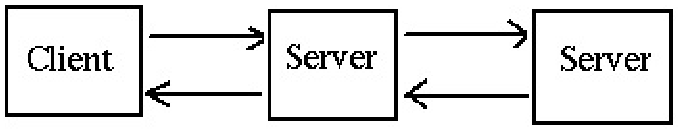


but you can also have multiple clients, single server



In this, the master receives requests and instead of handling them one at a time itself, passes them off to other servers to handle. This is a common model when concurrent clients are possible.

There are also single client, multiple servers



which occurs frequently when a server needs to act as a client to other servers, such as a business logic server getting information from a database server. And of course, there could be multiple clients with multiple servers.

Component Distribution

A simple but effective way of decomposing many applications is to consider them as made up of three parts:

- Presentation component
- Application logic
- Data access

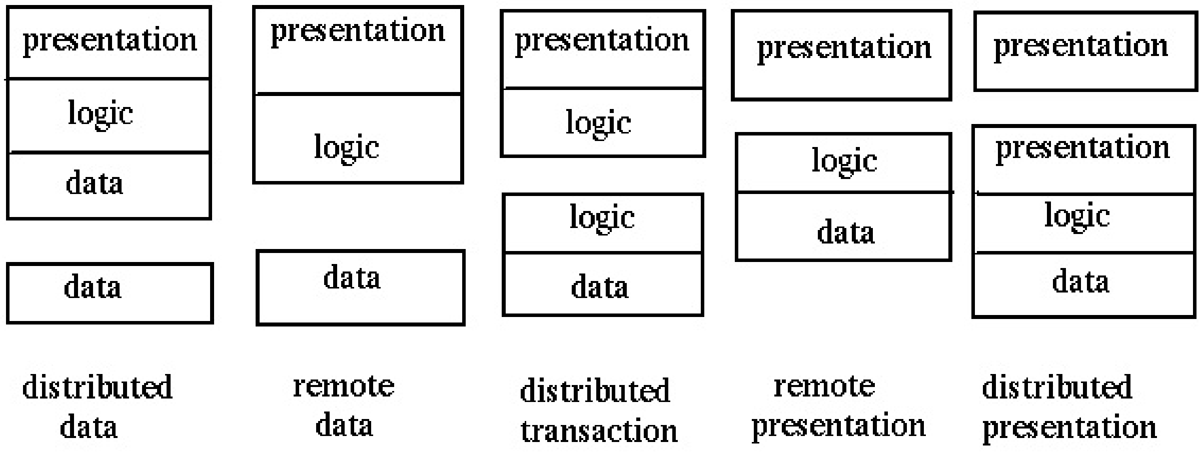
The *presentation component* is responsible for interactions with the user, both displaying data and gathering input. It may be a modern GUI interface with buttons, lists, menus, etc., or an older command-line style interface, asking questions and getting answers. The details are not important at this level.

The *application logic* is responsible for interpreting the users' responses, for applying business rules, for preparing queries and managing responses from the their component.

The *data access* component is responsible for storing and retrieving data. This will often be through a database, but not necessarily.

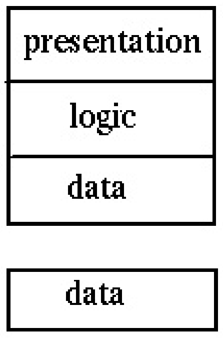
Gartner Classification

Based on this threefold decomposition of applications, Gartner considered how the components might be distributed in a client-server system. They came up with five models:



Example: Distributed Database

Gartner classification: 1

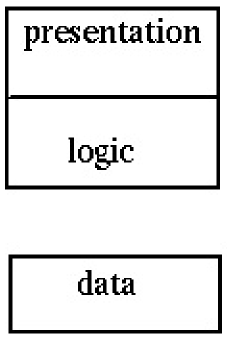


Modern mobile phones make good examples of this: due to limited memory they may store a small part of a database locally so that they can usual respond quickly. However, if data is required that is not held locally, then a request may be made to a remote database for that additional data.

Google maps forms another good example. All of the maps reside on Google's servers. When one is requested by a user, the "nearby" maps are also downloaded into a small database in the browser. When the user moves the map a little bit, the extra bits required are already in the local store for quick response.

Example: Network File Service

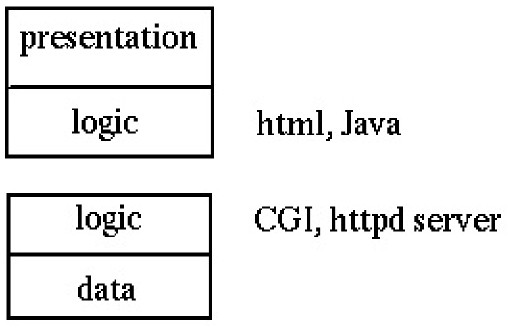
Gartner classification 2 allows remote clients access to a shared file system



There are many examples of such systems: NFS, Microsoft shares, DCE, etc.

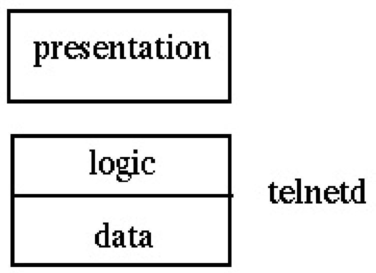
Example: Web

An example of Gartner classification 3 is the Web with Java applets. This is a distributed hypertext system, with many additional mechanisms.



Example: Terminal Emulation

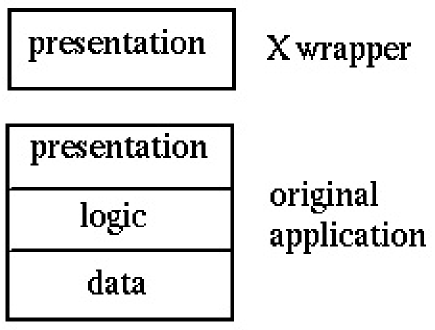
An example of Gartner classification 4 is terminal emulation. This allows a remote system to act as a normal terminal on a local system.



Telnet is the most common example of this.

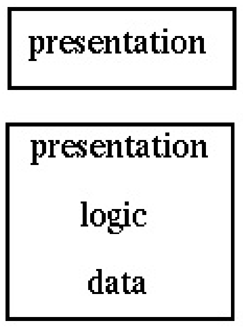
Example: Expect

Expect is a novel illustration of Gartner classification 5. It acts as a wrapper around a classical system such as a command-line interface. It builds an X Window interface around this, so that the user interacts with a GUI, and the GUI in turn interacts with the command-line interface.



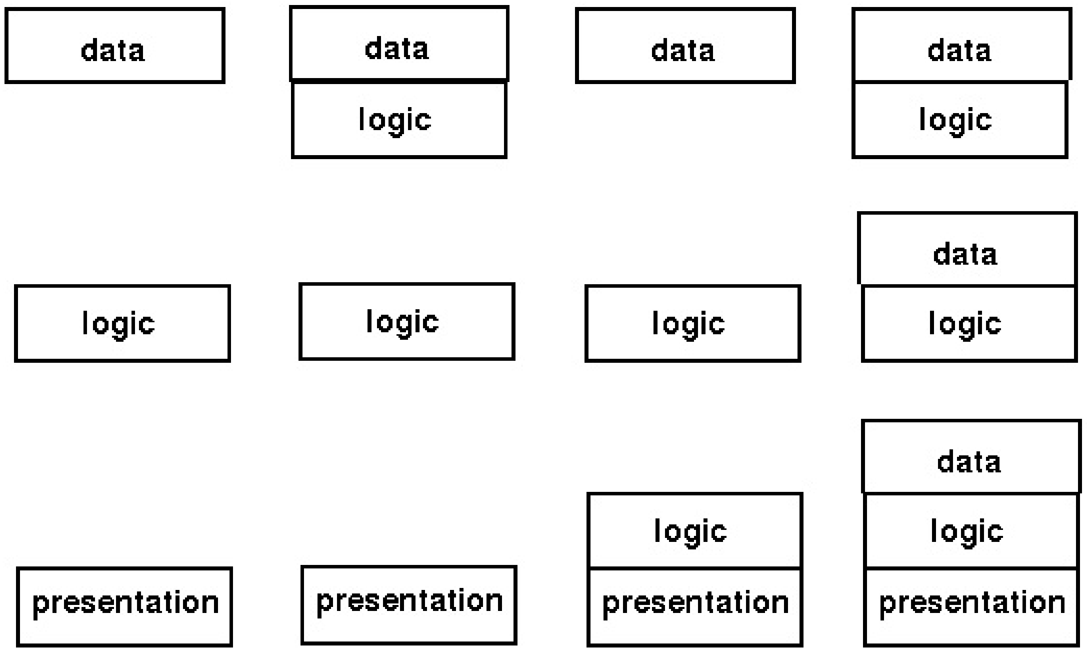
Example: X Window System

The X Window System itself is an example of Gartner classification 5. An application makes GUI calls such as *DrawLine*, but these are not handled directly but instead passed to an X Window server for rendering. This decouples the application view of windowing and the display view of windowing.



Three Tier Models

Of course, if you have two tiers, then you can have three, four, or more. Some of the three tier possibilities are shown in this diagram:



The modern Web is a good example of the rightmost of these. The backend is made up of a database, often running stored procedures to hold some of the database logic. The middle tier is an HTTP server such as Apache running PHP scripts (or Ruby on Rails, or JSP pages, etc). This will manage some of the logic and will have data such as HTML pages stored locally. The frontend is a browser to display the pages, under the control of some Javascript. In HTML 5, the frontend may also have a local database.

Fat vs thin

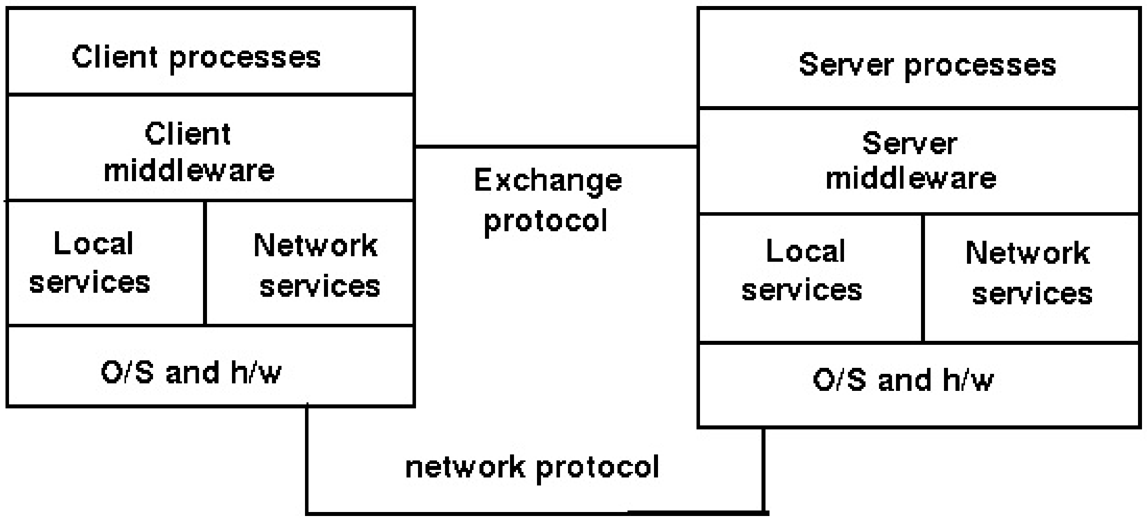
A common labelling of components is "fat" or "thin". Fat components take up lots of memory and do complex processing. Thin components on the other hand, do little of either. There don't seem to be any "normal" size components, only fat or thin!

Fatness or thinness is a relative concept. Browsers are often labelled as thin because "all they do is display web pages". Firefox on my Linux box takes nearly 1/2 a gigabyte of memory, which I don't regard as small at all!

Middleware

Middleware model

Middleware is tech "glue" connecting components of a distributed system. The middleware model is



Middleware

Components of middleware include

- The network services include things like TCP/IP
- The middleware layer is application-independent s/w using the network services.
- Examples of middleware are: DCE, RPC, Corba
- Middleware may only perform one function (such as RPC) or many (such as DCE)

Middleware examples

Examples of middleware include

- Primitive services such as terminal emulators, file transfer, email
- Basic services such as RPC
- Integrated services such as DCE, Network O/S
- Distributed object services such as CORBA, OLE/ActiveX
- Mobile object services such as RMI, Jini
- World Wide Web

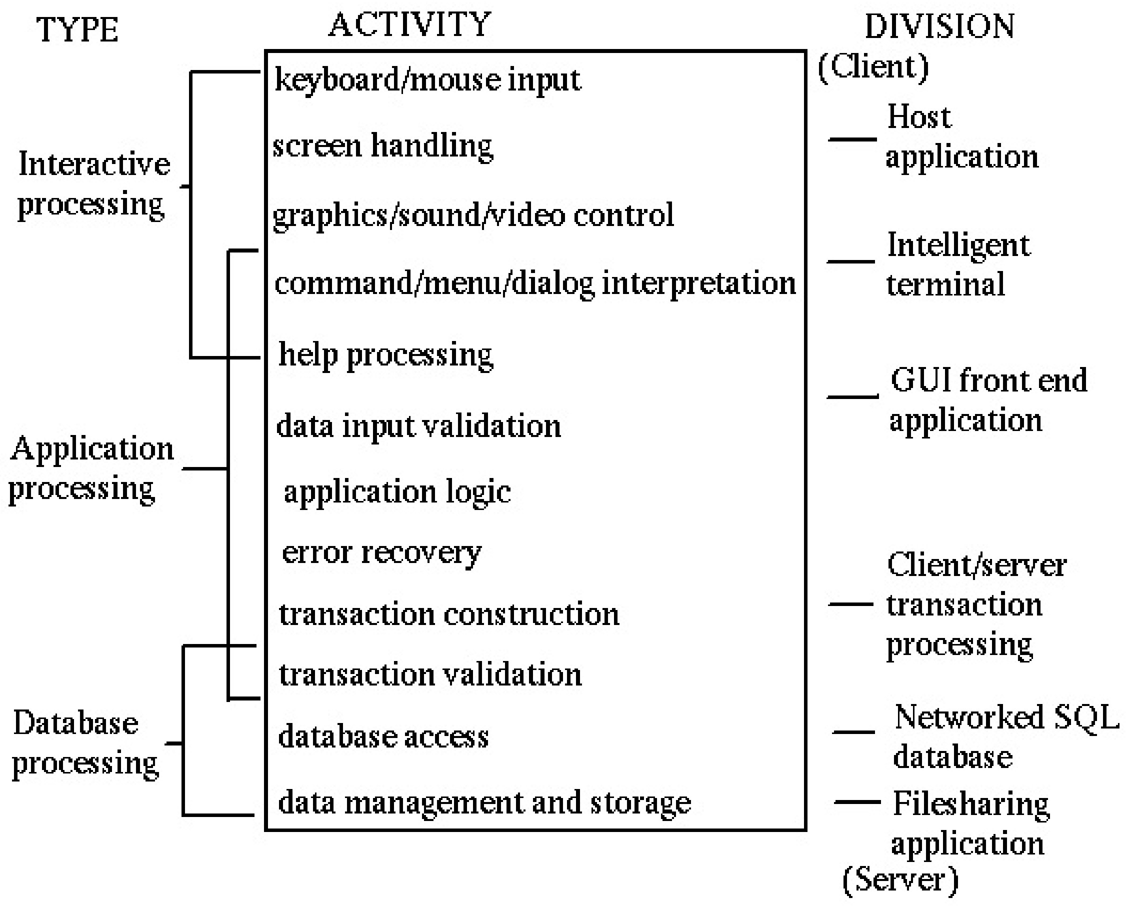
Middleware functions

The functions of middleware include

- Initiation of processes at different computers
- Session management
- Directory services to allow clients to locate servers
- Remote data access
- Concurrency control to allow servers to handle multiple clients
- Security and integrity
- Monitoring
- Termination of processes both local and remote

Continuum of Processing

The Gartner model is based on a breakdown of an application into the components of presentation, application logic and data handling. A finer grained breakdown is



Points of Failure

Distributed applications run in a complex environment. This makes them much more prone to failure than standalone applications on a single computer. The points of failure include

- The client side of the application could crash
- The client system may have h/w problems
- The client's network card could fail
- Network contention could cause timeouts
- There may be network address conflicts
- Network elements such as routers could fail
- Transmission errors may lose messages
- The client and server versions may be incompatible
- The server's network card could fail
- The server system may have h/w problems
- The server s/w may crash
- The server's database may become corrupted

Applications have to be designed with these possible failures in mind. Any action performed by one component must be recoverable if failure occurs in some other part of the system. Techniques such as transactions and continuous error checking need to be employed to avoid errors.

Acceptance Factors

- Reliability
- Performance
- Responsiveness
- Scalability
- Capacity
- Security

Transparency

The "holy grails" of distributed systems are to provide the following:

- access transparency
- location transparency
- migration transparency
- replication transparency
- concurrency transparency
- scalability transparency
- performance transparency
- failure transparency

Eight fallacies of distributed computing

Sun Microsystems was a company that performed much of the early work in distributed systems, and even had a mantra "The network is the computer." Based on their experience over many years a number of the scientists at Sun came up with the following list of fallacies commonly assumed:

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Many of these directly impact on network programming. For example, the design of most remote procedure call systems is based on the premise that the network is reliable so that a remote procedure call will behave in the same way as a local call. The fallacies of zero latency and infinite bandwidth also lead to assumptions about the time duration of an RPC call being the same as a local call, whereas they are orders of magnitude slower.

The recognition of these fallacies led Java's RMI (remote method invocation) model to require every RPC call to potentially throw a `RemoteException`. This forced programmers to at least recognise the possibility of network error and to remind them that they could not expect the same speeds as local calls.

Chapter 2 Overview of Go language

I don't feel like writing a chapter introducing Go right now, as there are other materials already available. There are several tutorials on the Go web site:

- Getting started
- A Tutorial for the Go Programming Language
- Effective Go

There is an introductory textbook on Go: "Go Programming" by John P. Baugh available from Amazon

Chapter 3 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 internet 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

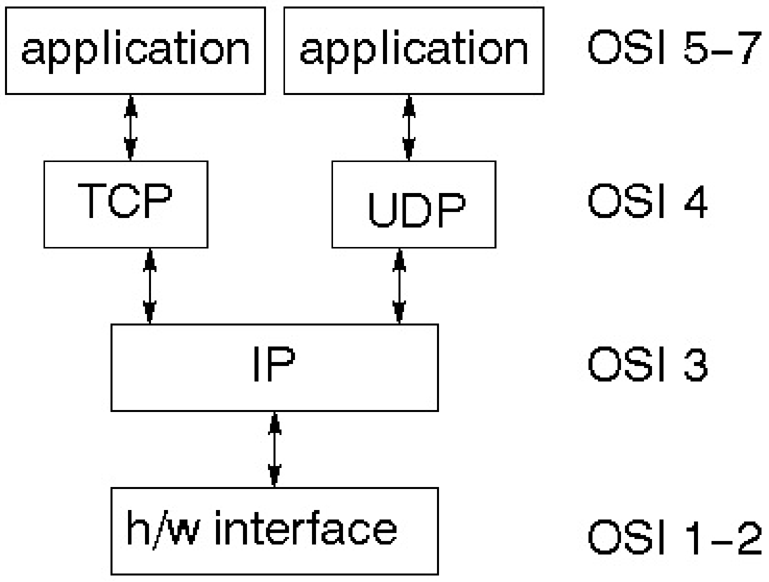
This chapter shows how to do TCP and UDP programming using Go, and how to use a raw socket for other protocols.

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 addresses

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 [1]
- 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 package `"net"` defines many types, functions and methods of use in Go network programming. The type `IP` is defined as byte slices

```
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) (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 between 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 successful, 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 on 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 := 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:

- There may be syntax errors in the address specified
- 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
- Although a connection has been established, writes to the service might fail if the connection has died suddenly, or the network times out
- 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 standardized 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("tcp4", 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 client can connect. Other clients are blocked, and will probably time out. Fortunately this is easily 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("tcp4", 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("udp4", 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 %s", 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("udp4", 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 %s", 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"
    "io/ioutil"
    "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)

    defer conn.Close()

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

    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)
    }
}
```

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 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 identifier
- 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.

```go
/* 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)
    }
}
```

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.

Chapter 4 Data serialisation

Communication between a client and a service requires the exchange of data. This data may be highly structured, but has to be serialised for transport. This chapter looks at the basics of serialisation and then considers several techniques supported by Go APIs.

Introduction

A client and server need to exchange information via messages. TCP and UDP provide the transport mechanisms to do this. The two processes also have to have a protocol in place so that message exchange can take place meaningfully.

Messages are sent across the network as a sequence of bytes, which has no structure except for a linear stream of bytes. We shall address the various possibilities for messages and the protocols that define them in the next chapter. In this chapter we concentrate on a component of messages - the data that is transferred.

A program will typically build complex data structures to hold the current program state. In conversing with a remote client or service, the program will be attempting to transfer such data structures across the network - that is, outside of the application's own address space.

Programming languages use structured data such as

- records/structures
- variant records
- array - fixed size or varying
- string - fixed size or varying
- tables - e.g. arrays of records
- non-linear structures such as
  - circular linked list
  - binary tree
  - objects with references to other objects

None of IP, TCP or UDP packets know the meaning of any of these data types. All that they can contain is a sequence of bytes. Thus an application has to *serialise* any data into a stream of bytes in order to write it, and *deserialise* the stream of bytes back into suitable data structures on reading it. These two operations are known as *marshalling* and *unmarshalling* respectively.

For example, consider sending the following variable length table of two columns of variable length strings:

fred	programmer
liping	analyst
sureerat	manager

This could be done by in various ways. For example, suppose that it is known that the data will be an unknown

This could be done by in various ways. For example, suppose that it is known that the data will be an unknown number of rows in a two-column table. Then a marshalled form could be

```
3                // 3 rows, 2 columns assumed
4 fred           // 4 char string,col 1
10 programmer    // 10 char string,col 2
6 liping         // 6 char string, col 1
7 analyst        // 7 char string, col 2
8 sureerat       // 8 char string, col 1
7 manager        // 7 char string, col 2
```

Variable length things can alternatively have their length indicated by terminating them with an "illegal" value, such as `'\0'` for strings:

```
3
fred\0
programmer\0
liping\0
analyst\0
sureerat\0
manager\0
```

Alternatively, it may be known that the data is a 3-row fixed table of two columns of strings of length 8 and 10 respectively. Then a serialisation could be

```
fred\0\0\0\0
programmer
liping\0\0
analyst\0\0\0
sureerat
manager\0\0\0
```

Any of these formats is okay - but the message exchange protocol must specify which one is used, or allow it to be determined at runtime.

Mutual agreement

The previous section gave an overview of the issue of data serialisation. In practise, the details can be considerably more complex. For example, consider the first possibility, marshalling a table into the stream

```
3
4 fred
10 programmer
6 liping
7 analyst
8 sureerat
7 manager
```

Many questions arise. For example, how many rows are possible for the table - that is, how big an integer do we need to describe the row size? If it is 255 or less, then a single byte will do, but if it is more, then a short, integer or long may be needed. A similar problem occurs for the length of each string. With the characters themselves, to which character set do they belong? 7 bit ASCII? 16 bit Unicode? The question of character sets is discussed at length in a later chapter.

The above serialisation is *opaque* or *implicit*. If data is marshalled using the above format, then there is nothing in the serialised data to say how it should be unmarshalled. The unmarshalling side has to know exactly how the data is serialised in order to unmarshal it correctly. For example, if the number of rows is marshalled as an eight-bit integer, but unmarshalled as a sixteen-bit integer, then an incorrect result will occur as the receiver tries to unmarshall 3 and 4 as a sixteen-bit integer, and the receiving program will almost certainly fail later.

An early well-known serialisation method is XDR (external data representation) used by Sun's RPC, later known as ONC (Open Network Computing). XDR is defined by RFC 1832 and it is instructive to see how precise this specification is. Even so, XDR is inherently type-unsafe as serialised data contains no type information. The correctness of its use in ONC is ensured primarily by compilers generating code for both marshalling and unmarshalling.

Go contains no explicit support for marshalling or unmarshalling opaque serialised data. The RPC package in Go does not use XDR, but instead uses "gob" serialisation, described later in this chapter.

Self-describing data

Self-describing data carries type information along with the data. For example, the previous data might get encoded as

```
table
    uint8 3
    uint 2
string
    uint8 4
    []byte fred
string
    uint8 10
    []byte programmer
string
    uint8 6
    []byte liping
string
    uint8 7
    []byte analyst
string
    uint8 8
    []byte sureerat
string
    uint8 7
    []byte manager
```

Of course, a real encoding would not normally be as cumbersome and verbose as in the example: small integers would be used as type markers and the whole data would be packed in as small a byte array as possible (XML provides a counter-example, though). However, the principle is that the marshaller will generate such type information in the serialised data. The unmarshaller will know the type-generation rules and will be able to use this to reconstruct the correct data structure.

ASN.1

Abstract Syntax Notation One (ASN.1) was originally designed in 1984 for the telecommunications industry. ASN.1 is a complex standard, and a subset of it is supported by Go in the package "asn1". It builds self-describing serialised data from complex data structures. Its primary use in current networking systems is as the encoding for X.509 certificates which are heavily used in authentication systems. The support in Go is based on what is needed to read and write X.509 certificates.

Two functions allow us to marshal and unmarshall data

```
func Marshal(val interface{}) ([]byte, os.Error)
func Unmarshal(val interface{}, b []byte) (rest []byte, err os.Error)
```

The first marshals a data value into a serialised byte array, and the second unmarshalls it. However, the first argument of type `interface` deserves further examination. Given a variable of a type, we can marshal it by just passing its value. To unmarshall it, we need a variable of a named type that will match the serialised data. The precise details of this are discussed later. But we also need to make sure that the variable is allocated to memory for that type, so that there is actually existing memory for the unmarshalling to write values into.

We illustrate with an almost trivial example, of marshalling and unmarshalling an integer. We can pass an integer value to `Marshal` to return a byte array, and unmarshall the array into an integer variable as in this program:

```
/* ASN.1
 */

package main

import (
    "encoding/asn1"
    "fmt"
    "os"
)

func main() {
    mdata, err := asn1.Marshal(13)
    checkError(err)

    var n int
    _, err1 := asn1.Unmarshal(mdata, &n)
    checkError(err1)

    fmt.Println("After marshal/unmarshal: ", n)
}

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

The unmarshalled value, is of course, 13.

Once we move beyond this, things get harder. In order to manage more complex data types, we have to look more closely at the data structures supported by ASN.1, and how ASN.1 support is done in Go.

Any serialisation method will be able to handle certain data types and not handle some others. So in order to determine the suitability of any serialisation such as ASN.1, you have to look at the possible data types supported versus those you wish to use in your application. The following ASN.1 types are taken from http://www.obj-sys.com/asn1tutorial/node4.html

The simple types are

- BOOLEAN: two-state variable values
- INTEGER: Model integer variable values
- BIT STRING: Model binary data of arbitrary length
- OCTET STRING: Model binary data whose length is a multiple of eight
- NULL: Indicate effective absence of a sequence element
- OBJECT IDENTIFIER: Name information objects
- REAL: Model real variable values
- ENUMERATED: Model values of variables with at least three states
- CHARACTER STRING: Models values that are strings of characters

Character strings can be from certain character sets

- NumericString: 0,1,2,3,4,5,6,7,8,9, and space
- PrintableString: Upper and lower case letters, digits, space, apostrophe, left/right parenthesis, plus sign, comma, hyphen, full stop, solidus, colon, equal sign, question mark
- TeletexString (T61String): The Teletex character set in CCITT's T61, space, and delete
- VideotexString: The Videotex character set in CCITT's T.100 and T.101, space, and delete
- VisibleString (ISO646String): Printing character sets of international ASCII, and space
- IA5String: International Alphabet 5 (International ASCII)
- GraphicString 25 All registered G sets, and space GraphicString

And finally, there are the structured types:

- SEQUENCE: Models an ordered collection of variables of different type
- SEQUENCE OF: Models an ordered collection of variables of the same type
- SET: Model an unordered collection of variables of different types
- SET OF: Model an unordered collection of variables of the same type
- CHOICE: Specify a collection of distinct types from which to choose one type
- SELECTION: Select a component type from a specified CHOICE type
- ANY: Enable an application to specify the type Note: ANY is a deprecated ASN.1 Structured Type. It has been replaced with X.680 Open Type.

Not all of these are supported by Go. Not all possible values are supported by Go. The rules as given in the Go "asn1" package documentation are

- An ASN.1 INTEGER can be written to an `int` or `int64` . If the encoded value does not fit in the Go type, Unmarshal returns a parse error.
- An ASN.1 BIT STRING can be written to a BitString.
- An ASN.1 OCTET STRING can be written to a `[]byte` .
- An ASN.1 OBJECT IDENTIFIER can be written to an ObjectIdentifier.
- An ASN.1 ENUMERATED can be written to an Enumerated.
- An ASN.1 UTCTIME or GENERALIZEDTIME can be written to a `*time.Time` .
- An ASN.1 PrintableString or IA5String can be written to a string.
- Any of the above ASN.1 values can be written to an `interface{}` . The value stored in the interface has the corresponding Go type. For integers, that type is `int64` .
- An ASN.1 SEQUENCE OF x or SET OF x can be written to a slice if an x can be written to * the slice's element type.
- An ASN.1 SEQUENCE or SET can be written to a struct if each of the elements in the * sequence can be written to the corresponding element in the struct.

Go places real restrictions on ASN.1. For example, ASN.1 allows integers of any size, while the Go implementation will only allow upto signed 64-bit integers. On the other hand, Go distinguishes between signed and unsigned types, while ASN.1 doesn't. So for example, transmitting a value of `uint64` may fail if it is too large for `int64` .

In a similar vein, ASN.1 allows several different character sets. Go only supports PrintableString and IA5String (ASCII). ASN.1 does not support Unicode characters (which require the BMPString ASN.1 extension). The basic Unicode character set of Go is not supported, and if an application requires transport of Unicode characters, then an encoding such as UTF-7 will be needed. Such encodings are discussed in a later chapter on character sets.

We have seen that a value such as an integer can be easily marshalled and unmarshalled. Other basic types such as booleans and reals can be similarly dealt with. Strings which are composed entirely of ASCII characters can be marshalled and unmarshalled. However, if the string is, for example, `"hello \u00bc"` which contains the non-ASCII character `'¼'` then an error will occur: `"ASN.1 structure error: PrintableString contains invalid character"` . This code works, as long as the string is only composed of printable characters:

```
s := "hello"
mdata, _ := asn1.Marshal(s)

var newstr string
asn1.Unmarshal(mdata, &newstr)
```

ASN.1 also includes some "useful types" not in the above list, such as UTC time. Go supports this UTC time type. This means that you can pass time values in a way that is not possible for other data values. ASN.1 does not support pointers, but Go has special code to manage pointers to time values. The function `GetLocalTime` returns `*time.Time` . The special code marshals this, and it can be unmarshalled into a pointer variable to a `time.Time` object. Thus this code works

```
t := time.LocalTime()
mdata, err := asn1.Marshal(t)

var newtime = new(time.Time)
_, err1 := asn1.Unmarshal(&newtime, mdata)
```

Both `LocalTime` and `new` handle pointers to a `*time.Time`, and Go looks after this special case.

In general, you will probably want to marshal and unmarshall structures. Apart from the special case of time, Go will happily deal with structures, but not with pointers to structures. Operations such as `new` create pointers, so you have to dereference them before marshalling/unmarshalling them. Go normally dereferences pointers for you when needed, but not in this case. These both work for a type `T`:

```
// using variables
var t1 T
t1 = ...
mdata1, _ := asn1.Marshal(t)

var newT1 T
asn1.Unmarshal(&newT1, mdata1)

/// using pointers
var t2 = new(T)
*t2 = ...
mdata2, _ := asn1.Marshal(*t2)

var newT2 = new(T)
asn1.Unmarshal(newT2, mdata2)
```

Any suitable mix of pointers and variables will work as well.

The fields of a structure must all be exportable, that is, field names must begin with an uppercase letter. Go uses the `reflect` package to marshall/unmarshall structures, so it must be able to examine all fields. This type cannot be marshalled:

```
type T struct {
  Field1 int
  field2 int // not exportable
}
```

ASN.1 only deals with the data types. It does not consider the names of structure fields. So the following type `T1` can be marshalled/unmarshalled into type `T2` as the corresponding fields are the same types:

```
type T1 struct {
    F1 int
    F2 string
}

type T2 struct {
    FF1 int
    FF2 string
}
```

Not only the types of each field must match, but the number must match as well. These two types don't work:

```
type T1 struct {
    F1 int
}

type T2 struct {
    F1 int
    F2 string // too many fields
}
```

ASN.1 daytime client and server

Now (finally) let us turn to using ASN.1 to transport data across the network.

We can write a TCP server that delivers the current time as an ASN.1 Time type, using the techniques of the last chapter. A server is

```
/* ASN1 DaytimeServer
 */
package main

import (
    "encoding/asn1"
    "fmt"
    "net"
    "os"
    "time"
)

func main() {

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

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

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

        daytime := time.Now()
        // Ignore return network errors.
        mdata, _ := asn1.Marshal(daytime)
        conn.Write(mdata)
        conn.Close() // we're finished
    }
}

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

which can be compiled to an executable such as `ASN1DaytimeServer` and run with no arguments. It will wait for connections and then send the time as an ASN.1 string to the client.

A client is

```
/* ASN.1 DaytimeClient
 */
package main

import (
    "bytes"
    "encoding/asn1"
    "fmt"
    "io"
    "io/ioutil"
    "net"
    "os"
    "time"
)

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)

    defer conn.Close()

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

    var newtime time.Time
    _, err1 := asn1.Unmarshal(result, &newtime)
    checkError(err1)

    fmt.Println("After marshal/unmarshal: ", newtime.String())

    os.Exit(0)
}

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

This connects to the service given in a form such as `localhost:1200`, reads the TCP packet and decodes the ASN.1 content back into a string, which it prints.

We should note that neither of these two - the client or the server - are compatible with the text-based clients and servers of the last chapter. This client and server are exchanging ASN.1 encoded data values, not textual strings.

JSON

JSON stands for JavaScript Object Notation. It was designed to be a lightweight means of passing data between JavaScript systems. It uses a text-based format and is sufficiently general that it has become used as a general purpose serialisation method for many programming languages.

JSON serialises objects, arrays and basic values. The basic values include string, number, boolean values and the null value. Arrays are a comma-separated list of values that can represent arrays, vectors, lists or sequences of various programming languages. They are delimited by square brackets `[ ... ]`. Objects are represented by a list of "field: value" pairs enclosed in curly braces `{ ... }`.

For example, the table of employees given earlier could be written as an array of employee objects:

```
[
   {Name: fred, Occupation: programmer},
   {Name: liping, Occupation: analyst},
   {Name: sureerat, Occupation: manager}
]
```

There is no special support for complex data types such as dates, no distinction between number types, no recursive types, etc. JSON is a very simple language, but nevertheless can be quite useful. Its text-based format makes it easy for people to use, even though it has the overheads of string handling.

From the Go JSON package specification, marshalling uses the following type-dependent default encodings:

- Boolean values encode as JSON booleans.
- Floating point and integer values encode as JSON numbers.
- String values encode as JSON strings, with each invalid UTF-8 sequence replaced by the encoding of the Unicode replacement character U+FFFD.
- Array and slice values encode as JSON arrays, except that []byte encodes as a base64-encoded string.
- Struct values encode as JSON objects. Each struct field becomes a member of the object. By default the object's key name is the struct field name converted to lower case. If the struct field has a tag, that tag will be used as the name instead.
- Map values encode as JSON objects. The map's key type must be string; the object keys are used directly as map keys.
- Pointer values encode as the value pointed to. (Note: this allows trees, but not graphs!). A nil pointer encodes as the null JSON object.
- Interface values encode as the value contained in the interface. A nil interface value encodes as the null JSON object.
- Channel, complex, and function values cannot be encoded in JSON. Attempting to encode such a value cause Marshal to return an `InvalidTypeError`.
- JSON cannot represent cyclic data structures and Marshal does not handle them. Passing cyclic structures to Marshal will result in an infinite recursion.

A program to store JSON serialised data into a file is

```
/* SaveJSON
 */

package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type Person struct {
    Name  Name
    Email []Email
}

type Name struct {
    Family   string
    Personal string
}

type Email struct {
    Kind    string
    Address string
}

func main() {
    person := Person{
        Name: Name{Family: "Newmarch", Personal: "Jan"},
        Email: []Email{Email{Kind: "home", Address: "jan@newmarch.name"},
            Email{Kind: "work", Address: "j.newmarch@boxhill.edu.au"}}}

    saveJSON("person.json", person)
}

func saveJSON(fileName string, key interface{}) {
    outFile, err := os.Create(fileName)
    checkError(err)
    encoder := json.NewEncoder(outFile)
    err = encoder.Encode(key)
    checkError(err)
    outFile.Close()
}

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

and to load it back into memory is

```
/* LoadJSON
 */
```

```go
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type Person struct {
    Name  Name
    Email []Email
}

type Name struct {
    Family   string
    Personal string
}

type Email struct {
    Kind    string
    Address string
}

func (p Person) String() string {
    s := p.Name.Personal + " " + p.Name.Family
    for _, v := range p.Email {
        s += "\n" + v.Kind + ": " + v.Address
    }
    return s
}
func main() {
    var person Person
    loadJSON("person.json", &person)

    fmt.Println("Person", person.String())
}

func loadJSON(fileName string, key interface{}) {
    inFile, err := os.Open(fileName)
    checkError(err)
    decoder := json.NewDecoder(inFile)
    err = decoder.Decode(key)
    checkError(err)
    inFile.Close()
}

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

The serialised form is (formatted nicely)

```
{"Name":{"Family":"Newmarch",
         "Personal":"Jan"},
 "Email":[{"Kind":"home","Address":"jan@newmarch.name"},
          {"Kind":"work","Address":"j.newmarch@boxhill.edu.au"}
         ]
}
```

A client and server

A client to send a person's data and read it back ten times is

```
/* JSON EchoClient
 */
package main

import (
    "fmt"
    "net"
    "os"
    "encoding/json"
    "bytes"
    "io"
)

type Person struct {
    Name  Name
    Email []Email
}

type Name struct {
    Family   string
    Personal string
}

type Email struct {
    Kind    string
    Address string
}

func (p Person) String() string {
    s := p.Name.Personal + " " + p.Name.Family
    for _, v := range p.Email {
        s += "\n" + v.Kind + ": " + v.Address
    }
    return s
}

func main() {
    person := Person{
        Name: Name{Family: "Newmarch", Personal: "Jan"},
        Email: []Email{Email{Kind: "home", Address: "jan@newmarch.name"},
            Email{Kind: "work", Address: "j.newmarch@boxhill.edu.au"}}}

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

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

    encoder := json.NewEncoder(conn)
    decoder := json.NewDecoder(conn)

    for n := 0; n < 10; n++ {
        encoder.Encode(person)
        var newPerson Person
        decoder.Decode(&newPerson)
        fmt.Println(newPerson.String())
    }

    os.Exit(0)
}

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

and the corresponding server is

```
/* JSON EchoServer
 */
package main

import (
    "fmt"
    "net"
    "os"
    "encoding/json"
)

type Person struct {
    Name  Name
    Email []Email
}

type Name struct {
    Family   string
    Personal string
}

type Email struct {
    Kind    string
    Address string
}
```

```
func (p Person) String() string {
    s := p.Name.Personal + " " + p.Name.Family
    for _, v := range p.Email {
        s += "\n" + v.Kind + ": " + v.Address
    }
    return s
}

func main() {

    service := "0.0.0.0:1200"
    tcpAddr, err := net.ResolveTCPAddr("tcp", service)
    checkError(err)

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

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

        encoder := json.NewEncoder(conn)
        decoder := json.NewDecoder(conn)

        for n := 0; n < 10; n++ {
            var person Person
            decoder.Decode(&person)
            fmt.Println(person.String())
            encoder.Encode(person)
        }
        conn.Close() // we're finished
    }
}

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

The gob package

Gob is a serialisation technique specific to Go. It is designed to encode Go data types specifically and does not at present have support for or by any other languages. It supports all Go data types except for channels and functions. It supports integers of all types and sizes, strings and booleans, structs, arrays and slices. At present it has some problems with circular structures such as rings, but that will improve over time.

Gob encodes type information into its serialised forms. This is far more extensive than the type information in say an X.509 serialisation, but far more efficient than the type information contained in an XML document. Type information is only included once for each piece of data, but includes, for example, the names of struct fields.

This inclusion of type information makes Gob marshalling and unmarshalling fairly robust to changes or differences between the marshaller and unmarshaller. For example, a struct

```
struct T {
    a int
    b int
}
```

can be marshalled and then unmarshalled into a different `struct`

```
struct T {
    b int
    a int
}
```

where the order of fields has changed. It can also cope with missing fields (the values are ignored) or extra fields (the fields are left unchanged). It can cope with pointer types, so that the above struct could be unmarshalled into

```
struct T {
    *a int
    **b int
}
```

To some extent it can cope with type coercions so that an `int` field can be broadened into an `int64`, but not with incompatible types such as `int` and `uint`.

To use Gob to marshall a data value, you first need to create an `Encoder`. This takes a `Writer` as parameter and marshalling will be done to this write stream. The encoder has a method `Encode` which marshalls the value to the stream. This method can be called multiple times on multiple pieces of data. Type information for each data type is only written once, though.

You use a `Decoder` to unmarshall the serialised data stream. This takes a `Reader` and each read returns an unmarshalled data value.

A program to store gob serialised data into a file is

```
/* SaveGob
 */

package main

import (
    "fmt"
    "os"
    "encoding/gob"
)

type Person struct {
    Name  Name
    Email []Email
}

type Name struct {
    Family   string
    Personal string
}

type Email struct {
    Kind    string
    Address string
}

func main() {
    person := Person{
        Name: Name{Family: "Newmarch", Personal: "Jan"},
        Email: []Email{Email{Kind: "home", Address: "jan@newmarch.name"},
            Email{Kind: "work", Address: "j.newmarch@boxhill.edu.au"}}}

    saveGob("person.gob", person)
}

func saveGob(fileName string, key interface{}) {
    outFile, err := os.Create(fileName)
    checkError(err)
    encoder := gob.NewEncoder(outFile)
    err = encoder.Encode(key)
    checkError(err)
    outFile.Close()
}

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

and to load it back into memory is

```
/* LoadGob
 */

package main

import (
    "fmt"
    "os"
    "encoding/gob"
)

type Person struct {
    Name  Name
    Email []Email
}

type Name struct {
    Family   string
    Personal string
}

type Email struct {
    Kind    string
    Address string
}

func (p Person) String() string {
    s := p.Name.Personal + " " + p.Name.Family
    for _, v := range p.Email {
        s += "\n" + v.Kind + ": " + v.Address
    }
    return s
}
func main() {
    var person Person
    loadGob("person.gob", &person)

    fmt.Println("Person", person.String())
}

func loadGob(fileName string, key interface{}) {
    inFile, err := os.Open(fileName)
    checkError(err)
    decoder := gob.NewDecoder(inFile)
    err = decoder.Decode(key)
    checkError(err)
    inFile.Close()
}

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

A client and server

A client to send a person's data and read it back ten times is

```
/* Gob EchoClient
 */
package main

import (
    "fmt"
    "net"
    "os"
    "encoding/gob"
    "bytes"
    "io"
)

type Person struct {
    Name  Name
    Email []Email
}

type Name struct {
    Family   string
    Personal string
}

type Email struct {
    Kind    string
    Address string
}

func (p Person) String() string {
    s := p.Name.Personal + " " + p.Name.Family
    for _, v := range p.Email {
        s += "\n" + v.Kind + ": " + v.Address
    }
    return s
}

func main() {
    person := Person{
        Name: Name{Family: "Newmarch", Personal: "Jan"},
        Email: []Email{Email{Kind: "home", Address: "jan@newmarch.name"},
            Email{Kind: "work", Address: "j.newmarch@boxhill.edu.au"}}}

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

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

    encoder := gob.NewEncoder(conn)
    decoder := gob.NewDecoder(conn)

    for n := 0; n < 10; n++ {
        encoder.Encode(person)
        var newPerson Person
        decoder.Decode(&newPerson)
        fmt.Println(newPerson.String())
    }

    os.Exit(0)
}

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

and the corrsponding server is

```
/* Gob EchoServer
 */
package main

import (
    "fmt"
    "net"
    "os"
    "encoding/gob"
)

type Person struct {
    Name  Name
    Email []Email
}

type Name struct {
    Family   string
    Personal string
}

type Email struct {
    Kind    string
    Address string
}

func (p Person) String() string {
    s := p.Name.Personal + " " + p.Name.Family
    for _, v := range p.Email {
        s += "\n" + v.Kind + ": " + v.Address
    }
    return s
}
```

```
func main() {

    service := "0.0.0.0:1200"
    tcpAddr, err := net.ResolveTCPAddr("tcp", service)
    checkError(err)

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

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

        encoder := gob.NewEncoder(conn)
        decoder := gob.NewDecoder(conn)

        for n := 0; n < 10; n++ {
            var person Person
            decoder.Decode(&person)
            fmt.Println(person.String())
            encoder.Encode(person)
        }
        conn.Close() // we're finished
    }
}

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

Encoding binary data as strings

Once upon a time, transmitting 8-bit data was problematic. It was often transmitted over noisy serial lines and could easily become corrupted. 7-bit data on the other hand could be transmitted more reliably because the 8th bit could be used as check digit. For example, in an "even parity" scheme, the check digit would be set to one or zero to make an even number of 1-s in a byte. This allows detection of errors of a single bit in each byte.

ASCII is a 7-bit character set. A number of schemes have been developed that are more sophisticated than simple parity checking, but which involve translating 8-bit binary data into 7-bit ASCII format. Essentially, the 8-bit data is stretched out in some way over the 7-bit bytes.

Binary data transmitted in HTTP responses and requests is often translated into an ASCII form. This makes it easy to inspect the HTTP messages with a simple text reader without worrying about what strange 8-bit bytes might do to your display!

One common format is Base64. Go has support for many binary-to-text formats, including Base64.

There are two principal functions to use for Base64 encoding and decoding:

```
func NewEncoder(enc *Encoding, w io.Writer) io.WriteCloser
func NewDecoder(enc *Encoding, r io.Reader) io.Reader
```

A simple program just to encode and decode a set of eight binary digits is

```
/**
 * Base64
 */

package main

import (
    "bytes"
    "encoding/base64"
    "fmt"
)

func main() {

    eightBitData := []byte{1, 2, 3, 4, 5, 6, 7, 8}
    bb := &bytes.Buffer{}
    encoder := base64.NewEncoder(base64.StdEncoding, bb)
    encoder.Write(eightBitData)
    encoder.Close()
    fmt.Println(bb)

    dbuf := make([]byte, 12)
    decoder := base64.NewDecoder(base64.StdEncoding, bb)
    decoder.Read(dbuf)
    for _, ch := range dbuf {
        fmt.Print(ch)
    }
}
```

Chapter 5 Application-Level Protocols

A client and a server exchange messages consisting of message types and message data. This requires design of a suitable message exchange protocol. This chapter looks at some of the issues involved in this, and gives a complete example of a simple client-server application.

Introduction

A client and server need to exchange information via messages. TCP and UDP provide the transport mechanisms to do this. The two processes also need to have a protocol in place so that message exchange can take place meaningfully. A protocol defines what type of conversation can take place between two components of a distributed application, by specifying messages, data types, encoding formats and so on.

Protocol Design

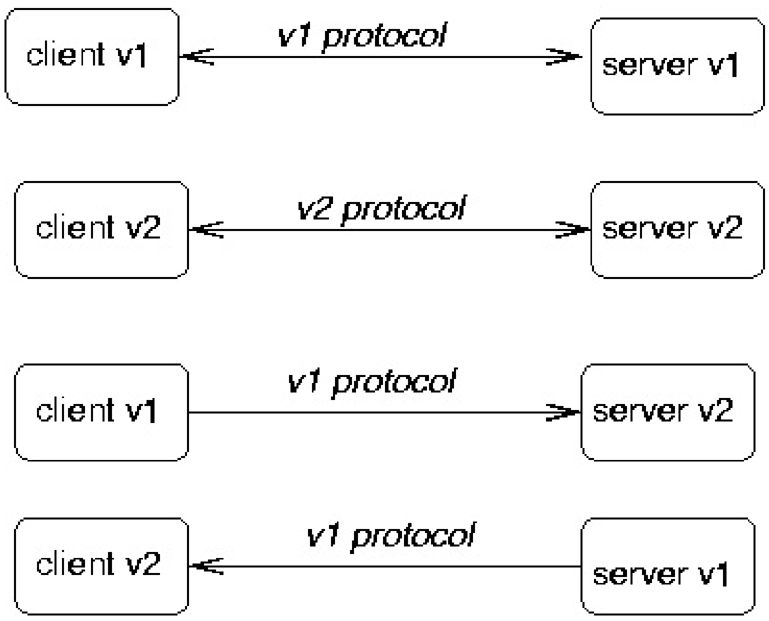
There are many possibilities and issues to be decided on when designing a protocol. Some of the issues include:

- Is it to be broadcast or point to point? Broadcast must be UDP, local multicast or the more experimental MBONE. Point to point could be either TCP or UDP.
- Is it to be stateful vs stateless? Is it reasonable for one side to maintain state about the other side? It is often simpler to do so, but what happens if something crashes?
- Is the transport protocol reliable or unreliable? Reliable is often slower, but then you don't have to worry so much about lost messages.
- Are replies needed? If a reply is needed, how do you handle a lost reply? Timeouts may be used.
- What data format do you want? Two common possibilities are MIME or byte encoding.
- Is your communication bursty or steady stream? Ethernet and the Internet are best at bursty traffic. Steady stream is needed for video streams and particularly for voice. If required, how do you manage Quality of Service (QoS)?
- Are there multiple streams with synchronisation required? Does the data need to be synchronised with anything, e.g. video and voice?
- Are you building a standalone application or a library to be used by others? The standards of documentation required might vary.

Version control

A protocol used in a client/server system will evolve over time, changing as the system expands. This raises compatibility problems: a version 2 client will make requests that a version 1 server doesn't understand, whereas a version 2 server will send replies that a version 1 client won't understand.

Each side should ideally be able to understand messages for its own version and all earlier ones. It should be able to write replies to old style queries in old style response format.



The ability to talk earlier version formats may be lost if the protocol changes too much. In this case, you need to be able to ensure that no copies of the earlier version still exist - and that is generally impossible.

Part of the protocol setup should involve version information.

The Web

The Web is a good example of a system that is messed up by different versions. The protocol has been through three versions, and most servers/browsers now use the latest version. The version is given in each request

request	version
GET /	pre 1.0
GET / HTTP/1.0	HTTP 1.0
GET / HTTP/1.1	HTTP 1.1

But the *content* of the messages has been through a large number of versions:

- HTML versions 1-4 (all different), with version 5 on the horizon;
- non-standard tags recognised by different browsers;
- non-HTML documents often require content handlers that may or may not be present - does your browser

have a handler for Flash?

- inconsistent treatment of document content (e.g. some stylesheet content will crash some browsers)
- Different support for JavaScript (and different versions of JavaScript)
- Different runtime engines for Java
- Many pages do not conform to *any* HTML versions (e.g. with syntax errors)

Message Format

In the last chapter we discussed some possibilities for representing data to be sent across the wire. Now we look one level up, to the messages which may contain such data.

- The client and server will exchange messages with different meanings, e.g.
    - login request,
    - get record request,
    - login reply,
    - record data reply.
- The client will prepare a request which must be understood by the server.
- The server will prepare a reply which must be understood by the client.

Commonly, the first part of the message will be a message type.

- Client to server

    ```
    LOGIN name passwd
    GET cpe4001 grade
    ```

- Server to client

    ```
    LOGIN succeeded
    GRADE cpe4001 D
    ```

The message types can be strings or integers. e.g. HTTP uses integers such as 404 to mean "not found" (although these integers are written as strings). The messages from client to server and vice versa are disjoint: "LOGIN" from client to server is different to "LOGIN" from server to client.

Data Format

There are two main format choices for messages: byte encoded or character encoded.

Byte format

In the byte format

- the first part of the message is typically a byte to distinguish between message types.
- The message handler would examine this first byte to distinguish message type and then perform a switch to select the appropriate handler for that type.
- Further bytes in the message would contain message content according to a pre-defined format (as discussed in the previous chapter).

The advantages are compactness and hence speed. The disadvantages are caused by the opaqueness of the data: it may be harder to spot errors, harder to debug, require special purpose decoding functions. There are many examples of byte-encoded formats, including major protocols such as DNS and NFS, upto recent ones such as Skype. Of course, if your protocol is not publicly specified, then a byte format can also make it harder for others to reverse-engineer it!

Pseudocode for a byte-format server is

```
handleClient(conn) {
    while (true) {
        byte b = conn.readByte()
        switch (b) {
            case MSG_1: ...
            case MSG_2: ...
            ...
        }
    }
}
```

Go has basic support for managing byte streams. The interface `Conn` has methods

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

and these methods are implemented by `TCPConn` and `UDPConn` .

Character Format

In this mode, everything is sent as characters if possible. For example, an integer 234 would be sent as, say, the three characters '2', '3' and '4' instead of the one byte 234. Data that is inherently binary may be base64 encoded to change it into a 7-bit format and then sent as ASCII characters, as discussed in the previous chapter.

In character format

In character format,

- A message is a sequence of one or more lines The start of the first line of the message is typically a word that represents the message type.
- String handling functions may be used to decode the message type and data.
- The rest of the first line and successive lines contain the data.
- Line-oriented functions and line-oriented conventions are used to manage this.

Pseudocode is

```
handleClient() {
    line = conn.readLine()
    if (line.startsWith(...) {
        ...
    } else if (line.startsWith(...) {
        ...
    }
}
```

Character formats are easier to setup and easier to debug. For example, you can use *telnet* to connect to a server on any port, and send client requests to that server. It isn't so easy the other way, but you can use tools like *tcpdump* to snoop on TCP traffic and see immediately what clients are sending to servers.

There is not the same level of support in Go for managing character streams. There are significant issues with character sets and character encodings, and we will explore these issues in a later chapter.

If we just pretend everything is ASCII, like it was once upon a time, then character formats are quite straightforward to deal with. The principal complication at this level is the varying status of *"newline"* across different operating systems. Unix uses the single character `"\n"`. Windows and others (more correctly) use the pair `"\r\n"`. On the internet, the pair `"\r\n"` is most common - Unix systems just need to take care that they don't assume `"\n"`.

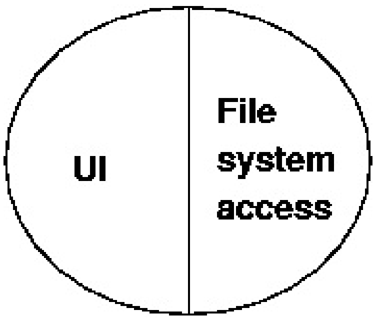
Simple Example

This example deals with a directory browsing protocol - basically a stripped down version of FTP, but without even the file transfer part. We only consider listing a directory name, listing the contents of a directory and changing the current directory - all on the server side, of course. This is a complete worked example of creating all components of a client-server application. It is a simple program which includes messages in both directions, as well as design of messaging protocol.

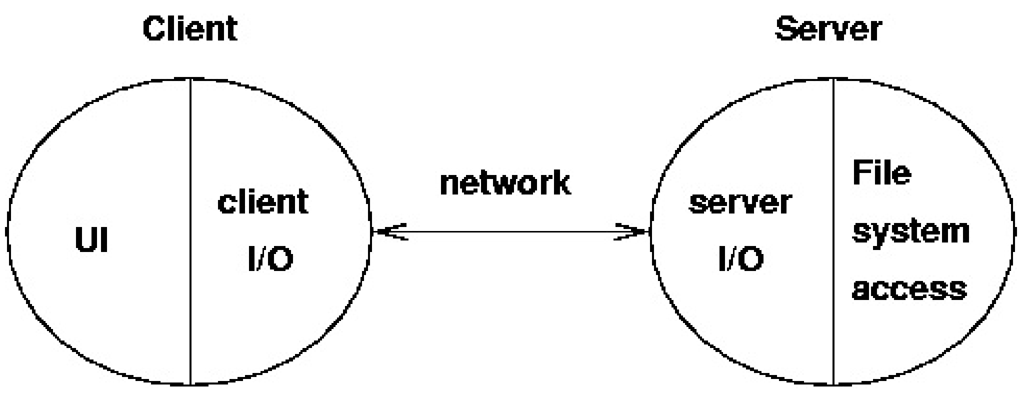
Look at a simple non-client-server program that allows you to list files in a directory and change and print the directory on the server. We omit copying files, as that adds to the length of the program without really introducing important concepts. For simplicity, all filenames will be assumed to be in 7-bit ASCII. If we just looked at a standalone application first, then the pseudo-code would be

```
read line from user
while not eof do
  if line == dir
    list directory
  else

  if line == cd <dir>
    change directory
  else

  if line == pwd
    print directory
  else

  if line == quit
    quit
  else
    complain

  read line from user
```

A non-distributed application would just link the UI and file access code



In a client-server situation, the client would be at the user end, talking to a server somewhere else. Aspects of this program belong solely at the presentation end, such as getting the commands from the user. Some are messages from the client to the server, some are solely at the server end.



For a simple directory browser, assume that all directories and files are at the server end, and we are only transferring file information from the server to the client. The client side (including presentation aspects) will become

```
read line from user
while not eof do
  if line == dir
    *list directory*
  else

  if line == cd <dir>
    *change directory*
  else

  if line == pwd
    *print directory*
  else

  if line == quit
    quit
  else
    complain

  read line from user
```

where the stared lines involve communication with the server.

Alternative presentation aspects

A GUI program would allow directory contents to be displayed as lists, for files to be selected and actions such as change directory to be be performed on them. The client would be controlled by actions associated with various events that take place in graphical objects. The pseudo-code might look like

```
change dir button:
  if there is a selected file
    *change directory*
  if successful
    update directory label
    *list directory*
    update directory list
```

The functions called from the different UI's should be the same - changing the presentation should not change the networking code

Protocol - informal

client request	server response
`dir`	send list of files
`cd <dir>`	change dir send error if failed send ok if succeed
`pwd`	send current directory
`quit`	quit

Text protocol

This is a simple protocol. The most complicated data structure that we need to send is an array of strings for a directory listing. In this case we don't need the heavy duty serialisation techniques of the last chapter. In this case we can use a simple text format.

But even if we make the protocol simple, we still have to specify it in detail. We choose the following message format:

- All messages are in 7-bit US-ASCII
- The messages are case-sensitive
- Each message consists of a sequence of lines
- The first word on the first line of each message describes the message type. All other words are message data
- All words are separated by exactly one space character
- Each line is terminated by CR-LF

Some of the choices made above are weaker in real-life protocols. For example

- Message types could be case-insensitive. This just requires mapping message type strings down to lower-case before decoding
- An arbitrary amount of white space could be left between words. This just adds a little more complication, compressing white space
- Continuation characters such as `"\"` can be used to break long lines over several lines. This starts to make processing more complex
- Just a `"\n"` could be used as line terminator, as well as `"\r\n"`. This makes recognising end of line a bit harder

All of these variations exist in real protocols. Cumulatively, they make the string processing just more complex than in our case.

client request	server response
[illegible]	[illegible]

send `"DIR"`	send list of files, one per line terminated by a blank line
send `"CD <dir>"`	change dir send "ERROR" if failed send "OK"
send `"PWD"`	send current working directory

Server code

```
/* FTP Server
 */
package main

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

const (
    DIR = "DIR"
    CD  = "CD"
    PWD = "PWD"
)

func main() {

    service := "0.0.0.0:1202"
    tcpAddr, err := net.ResolveTCPAddr("tcp", service)
    checkError(err)

    listener, err := net.ListenTCP("tcp", tcpAddr)
    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 {
            conn.Close()
            return
        }
        s := string(buf[0:n])
        // decode request
        if s[0:2] == CD {
            chdir(conn, s[3:])
        } else if s[0:3] == DIR {
            dirList(conn)
        } else if s[0:3] == PWD {
            pwd(conn)
        }

    }
}

func chdir(conn net.Conn, s string) {
    if os.Chdir(s) == nil {
        conn.Write([]byte("OK"))
    } else {
        conn.Write([]byte("ERROR"))
    }
}

func pwd(conn net.Conn) {
    s, err := os.Getwd()
    if err != nil {
        conn.Write([]byte(""))
        return
    }
    conn.Write([]byte(s))
}

func dirList(conn net.Conn) {
    defer conn.Write([]byte("\r\n"))

    dir, err := os.Open(".")
    if err != nil {
        return
    }

    names, err := dir.Readdirnames(-1)
    if err != nil {
        return
    }
    for _, nm := range names {
        conn.Write([]byte(nm + "\r\n"))
    }
}

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

Client code

```
/* FTPClient
 */
package main

import (
    "fmt"
    "net"
    "os"
    "bufio"
    "strings"
    "bytes"
)

// strings used by the user interface
const (
    uiDir  = "dir"
    uiCd   = "cd"
    uiPwd  = "pwd"
    uiQuit = "quit"
)

// strings used across the network
const (
    DIR = "DIR"
    CD  = "CD"
    PWD = "PWD"
)

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

    host := os.Args[1]

    conn, err := net.Dial("tcp", host+":1202")
    checkError(err)

    reader := bufio.NewReader(os.Stdin)
    for {
        line, err := reader.ReadString('\n')
        // lose trailing whitespace
        line = strings.TrimRight(line, " \t\r\n")
        if err != nil {
            break
        }

        // split into command + arg
        strs := strings.SplitN(line, " ", 2)
        // decode user request
        switch strs[0] {
        case uiDir:
            dirRequest(conn)
        case uiCd:
            if len(strs) != 2 {
                fmt.Println("cd <dir>")
                continue
            }
            fmt.Println("CD \"", strs[1], "\"")
            cdRequest(conn, strs[1])
        case uiPwd:
            pwdRequest(conn)
        case uiQuit:
            conn.Close()
            os.Exit(0)
        default:
            fmt.Println("Unknown command")
        }
    }
}

func dirRequest(conn net.Conn) {
    conn.Write([]byte(DIR + " "))

    var buf [512]byte
    result := bytes.NewBuffer(nil)
    for {
        // read till we hit a blank line
        n, _ := conn.Read(buf[0:])
        result.Write(buf[0:n])
        length := result.Len()
        contents := result.Bytes()
        if string(contents[length-4:]) == "\r\n\r\n" {
            fmt.Println(string(contents[0 : length-4]))
            return
        }
    }
}

func cdRequest(conn net.Conn, dir string) {
    conn.Write([]byte(CD + " " + dir))
    var response [512]byte
    n, _ := conn.Read(response[0:])
    s := string(response[0:n])
    if s != "OK" {
        fmt.Println("Failed to change dir")
    }
}

func pwdRequest(conn net.Conn) {
    conn.Write([]byte(PWD))
    var response [512]byte
    n, _ := conn.Read(response[0:])
    s := string(response[0:n])
    fmt.Println("Current dir \"" + s + "\"")
}

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

State

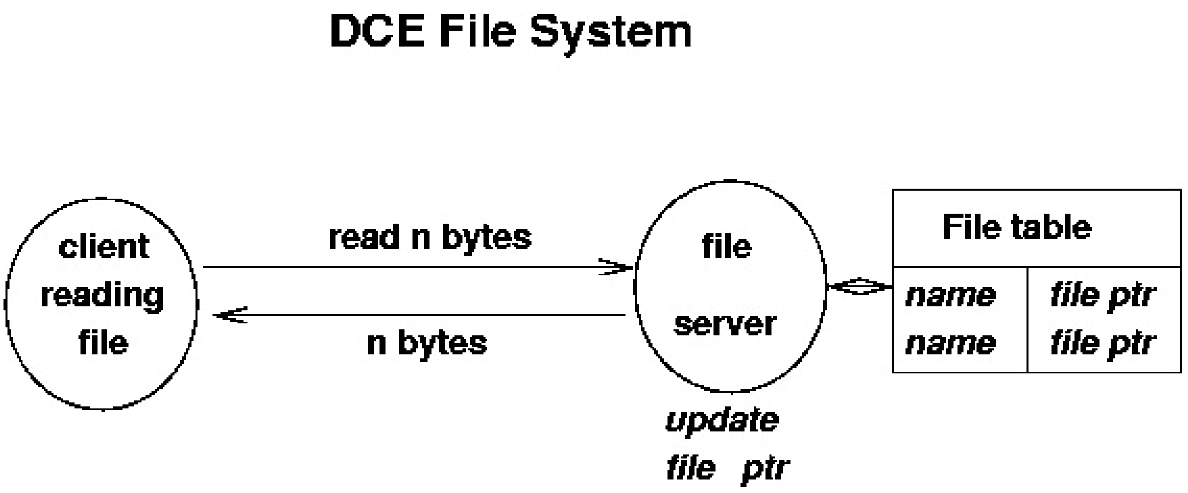
Applications often make use of state information to simplify what is going on. For example

- Keeping file pointers to current file location
- Keeping current mouse position
- Keeping current customer value

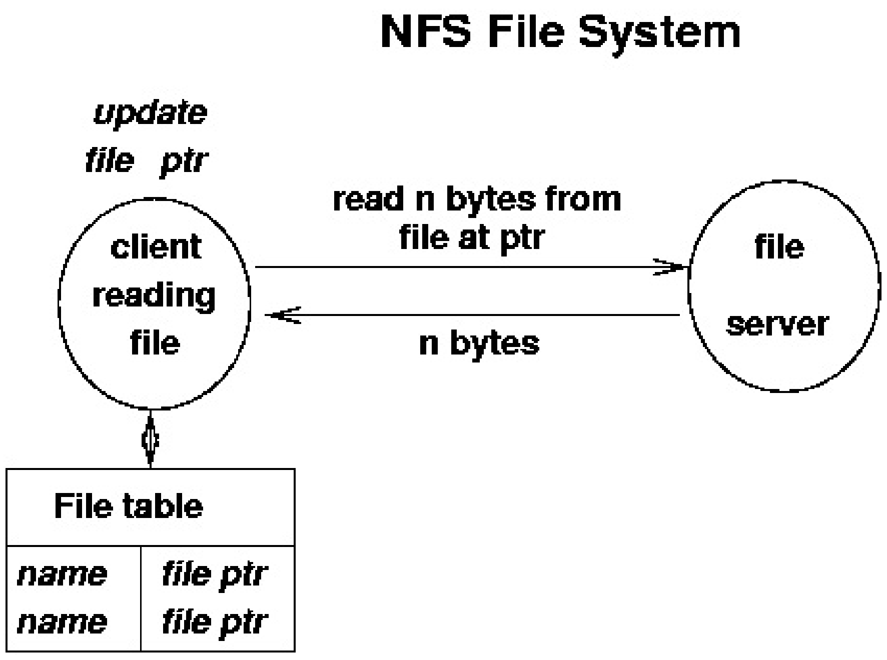
In a distributed system, such state information may be kept in the client, in the server, or in both.

The important point is to whether one process is keeping state information about itself or about the other process. One process may keep as much state information about itself as it wants, without causing any problems. If it needs to keep information about the state of the other process, then problems arise: the process' actual knowledge of the state of the other may become incorrect. This can be caused by loss of messages (in UDP), by failure to update, or by s/w errors.

An example is reading a file. In single process applications the file handling code runs as part of the application. It maintains a table of open files and the location in each of them. Each time a read or write is done this file location is updated. In the DCE file system, the file server keeps track of a client's open files, and where the client's file pointer is. If a message could get lost (but DCE uses TCP) these could get out of synch. If the client crashes, the server must eventually timeout on the client's file tables and remove them.



In NFS, the server does not maintain this state. The client does. Each file access from the client that reaches the server must open the file at the appropriate point, as given by the client, to perform the action.



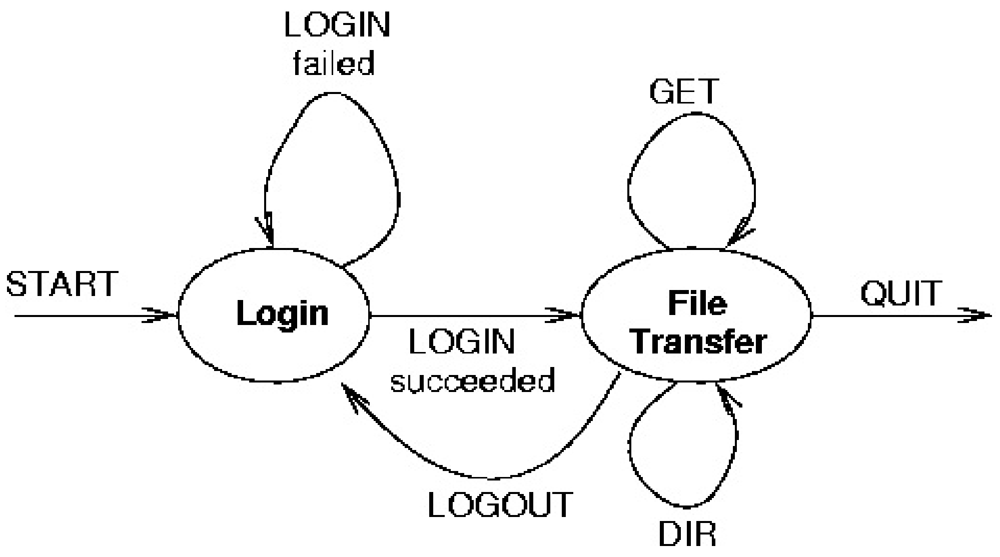
If the server maintains information about the client, then it must be able to recover if the client crashes. If information is not saved, then on each transaction the client must transfer sufficient information for the server to function.

If the connection is unreliable, then additional handling must be in place to ensure that the two do not get out of synch. The classic example is of bank account transactions where the messages get lost. A transaction server may need to be part of the client-server system.

Application State Transition Diagram

A state transition diagram keeps track of the current state of an application and the changes that move it to new states.

Example: file transfer with login:



This can also be expressed as a table

Current state	Transition	Next state
login	login failed	login
	login succeeded	file transfer
file transfer	dir	file transfer
	get	file transfer
	logout	login
	quit	-

Client state transition diagrams

The client state diagram must follow the application diagram. It has more detail though: it *writes* and then *reads*

Current state	Write	Read	Next state
login	LOGIN name password	FAILED	login
		SUCCEEDED	file transfer
file transfer	CD dir	SUCCEEDED	file transfer
		FAILED	file transfer
	GET filename	#lines + contents	file transfer
		ERROR	file transfer
	DIR	#files + filenames	file transfer
		ERROR	file transfer
	quit	none	quit
logout	none	login	

Server state transition diagrams

The server state diagram must also follow the application diagram. It also has more detail: it *reads* and then *writes*

Current state	Write	Read	Next state
login	LOGIN name password	FAILED	login
		SUCCEEDED	file transfer
file transfer	CD dir	SUCCEEDED	file transfer
		FAILED	file transfer
	GET filename	#lines + contents	file transfer
		ERROR	file transfer

	DIR	#files + filenames	file transfer
		ERROR	file transfer
	quit	none	quit
logout	none	login	

Server pseudocode

```
state = login
while true
    read line
    switch (state)
        case login:
            get NAME from line
            get PASSWORD from line
            if NAME and PASSWORD verified
                write SUCCEEDED
                state = file_transfer
            else
                write FAILED
                state = login
        case file_transfer:
            if line.startsWith CD
                get DIR from line
                if chdir DIR okay
                    write SUCCEEDED
                    state = file_transfer
                else
                    write FAILED
                    state = file_transfer
            ...
```

We don't give the actual code for this server or client since it is pretty straightforward.

Conclusion

Building any application requires design decisions before you start writing code. For distributed applications you have a wider range of decisions to make compared to standalone systems. This chapter has considered some of those aspects and demonstrated what the resultant code might look like.

Chapter 6 Managing character sets and encodings

There are many languages in use throughout the world, and they use many different character sets. There are also many ways of encoding character sets into binary formats of bytes. This chapter considers some of the issues in this.

Introduction

Once upon a time there was EBCDIC and ASCII... Actually, it was never that simple and has just become more complex over time. There is light on the horizon, but some estimates are that it may be 50 years before we all live in the daylight on this!

Early computers were developed in the English speaking countries of the US, the UK and Australia. As a result of this, assumptions were made about the language and character sets in use. Basically, the Latin alphabet was used, plus numerals, punctuation characters and a few others. These were then encoded into bytes using ASCII or EBCDIC.

The character-handling mechanisms were based on this: text files and I/O consisted of a sequence of bytes, with each byte representing a single character. String comparison could be done by matching corresponding bytes; conversions from upper to lower case could be done by mapping individual bytes, and so on.

There are about 6,000 living languages in the world (3,000 of them in Papua New Guinea!). A few languages use the "english" characters but most do not. The Romanic languages such as French have adornments on various characters, so that you can write "j'ai arrêté", with two differently accented vowels. Similarly, the Germanic languages have extra characters such as 'ß'. Even UK English has characters not in the standard ASCII set: the pound symbol '£' and recently the euro '€'

But the world is not restricted to variations on the Latin alphabet. Thailand has its own alphabet, with words looking like this: "□□□□□□□". There are many other alphabets, and Japan even has two, Hiragana and Katagana.

There are also the hieroglyphic languages such as Chinese where you can write "百度一下，你就知道".

It would be nice from a technical viewpoint if the world just used ASCII. However, the trend is in the opposite direction, with more and more users demanding that software use the language that they are familiar with. If you build an application that can be run in different countries then users will demand that it uses their own language. In a distributed system, different components of the system may be used by users expecting different languages and characters.

Internationalisation (i18n) is how you write your applications so that they can handle the variety of languages and cultures. *Localisation* (l10n) is the process of customising your internationalised application to a particular cultural group.

i18n and l10n are big topics in themselves. For example, they cover issues such as colours: while white means "purity" in Western cultures, it means "death" to the Chinese and "joy" to Egyptians. In this chapter we just look at issues of character handling.

Definitions

It is important to be careful about exactly what part of a text handling system you are talking about. Here is a set of definitions that have proven useful.

Character

A character is a "unit of information that roughly corresponds to a grapheme (written symbol) of a natural language, such as a letter, numeral, or punctuation mark" (Wikipedia). A character is "the smallest component of written language that has a semantic value" (Unicode). This includes letters such as 'a' and 'À' (or letters in any other language), digits such as '2', punctuation characters such as ',' and various symbols such as the English pound currency symbol '£'.

A character is some sort of abstraction of any actual symbol: the character 'a' is to any written 'a' as a Platonic circle is to any actual circle. The concept of character also includes control characters, which do not correspond to natural language symbols but to other bits of information used to process texts of the language.

A character does not have any particular appearance, although we use the appearance to help recognise the character. However, even the appearance may have to be understood in a context: in mathematics, if you see the symbol π (pi) it is the character for the ratio of circumference to radius of a circle, while if you are reading Greek text, it is the sixteenth letter of the alphabet: "προσ" is the greek word for "with" and has nothing to do with 3.14159...

Character repertoire/character set

A character repertoire is a set of distinct characters, such as the Latin alphabet. No particular ordering is assumed. In English, although we say that 'a' is earlier in the alphabet than 'z', we wouldn't say that 'a' is less than 'z'. The "phone book" ordering which puts "McPhee" before "MacRea" shows that "alphabetic ordering" isn't critical to the characters.

A repertoire specifies the names of the characters and often a sample of how the characters might look. e.g the letter 'a' might look like 'a', 'a' or 'a'. But it doesn't force them to look like that - they are just samples. The repertoire may make distinctions such as upper and lower case, so that 'a' and 'A' are different. But it may regard them as the same, just with different sample appearances. (Just like some programming languages treat upper and lower as different - e.g. Go - but some don't e.g. Basic.). On the other hand, a repertoire might contain different characters with the same sample appearance: the repertoire for a Greek mathematician would have two different characters with appearance π. This is also called a noncoded character set.

Character code

A character code is a mapping from characters to integers. The mapping for a character set is also called a coded character set or code set. The value of each character in this mapping is often called a code point. ASCII is a code set. The codepoint for 'a' is 97 and for 'A' is 65 (decimal).

The character code is still an abstraction. It isn't yet what we will see in text files, or in TCP packets. However, The character code is still an abstraction. It isn't yet what we will see in text files, or in TCP packets. However, it is getting close. as it supplies the mapping from human oriented concepts into numerical ones.

Character encoding

To communicate or store a character you need to encode it in some way. To transmit a string, you need to encode all characters in the string. There are many possible encodings for any code set.

For example, 7-bit ASCII code points can be encoded as themselves into 8-bit bytes (an octet). So ASCII 'A' (with codepoint 65) is encoded as the 8-bit octet 01000001. However, a different encoding would be to use the top bit for parity checking e.g. with odd parity ASCII 'A" would be the octet 11000001. Some protocols such as Sun's XDR use 32-bit word-length encoding. ASCII 'A' would be encoded as 00000000 00000000 0000000 01000001.

The character encoding is where we function at the programming level. Our programs deal with encoded characters. It obviously makes a difference whether we are dealing with 8-bit characters with or without parity checking, or with 32-bit characters.

The encoding extends to strings of characters. A word-length even parity encoding of "ABC" might be 10000000 (parity bit in high byte) 0100000011 (C) 01000010 (B) 01000001 (A in low byte). The comments about the importance of an encoding apply equally strongly to strings, where the rules may be different.

Transport encoding

A character encoding will suffice for handling characters within a single application. However, once you start sending text *between* applications, then there is the further issue of how the bytes, shorts or words are put on the wire. An encoding can be based on space- and hence bandwidth-saving techniques such as `zip` 'ping the text. Or it could be reduced to a 7-bit format to allow a parity checking bit, such as `base64` .

If we do know the character and transport encoding, then it is a matter of programming to manage characters and strings. If we don't know the character or transport encoding then it is a matter of guesswork as to what to do with any particular string. There is no convention for files to signal the character encoding.

There is however a convention for signalling encoding in text transmitted across the internet. It is simple: the header of a text message contains information about the encoding. For example, an HTTP header can contain lines such as

```
Content-Type: text/html; charset=ISO-8859-4
Content-Encoding: gzip
```

which says that the character set is ISO 8859-4 (corresponding to certain countries in Europe) with the default encoding, but then `gzip` ed. The second part - content encoding - is what we are referring to as "transfer encoding" (IETF RFC 2130).

But how do you read this information? Isn't it encoded? Don't we have a chicken and egg situation? Well, no. The convention is that such information is given in ASCII (to be precise, US ASCII) so that a program can read the headers and then adjust its encoding for the rest of the document.

ASCII

ASCII has the repertoire of the English characters plus digits, punctuation and some control characters. The code points for ASCII are given by the familiar table

```
Oct   Dec   Hex   Char           Oct   Dec   Hex   Char
------------------------------------------------------------
000   0     00    NUL '\0'       100   64    40    @
001   1     01    SOH            101   65    41    A
002   2     02    STX            102   66    42    B
003   3     03    ETX            103   67    43    C
004   4     04    EOT            104   68    44    D
005   5     05    ENQ            105   69    45    E
006   6     06    ACK            106   70    46    F
007   7     07    BEL '\a'       107   71    47    G
010   8     08    BS  '\b'       110   72    48    H
011   9     09    HT  '\t'       111   73    49    I
012   10    0A    LF  '\n'       112   74    4A    J
013   11    0B    VT  '\v'       113   75    4B    K
014   12    0C    FF  '\f'       114   76    4C    L
015   13    0D    CR  '\r'       115   77    4D    M
016   14    0E    SO             116   78    4E    N
017   15    0F    SI             117   79    4F    O
020   16    10    DLE            120   80    50    P
021   17    11    DC1            121   81    51    Q
022   18    12    DC2            122   82    52    R
023   19    13    DC3            123   83    53    S
024   20    14    DC4            124   84    54    T
025   21    15    NAK            125   85    55    U
026   22    16    SYN            126   86    56    V
027   23    17    ETB            127   87    57    W
030   24    18    CAN            130   88    58    X
031   25    19    EM             131   89    59    Y
032   26    1A    SUB            132   90    5A    Z
033   27    1B    ESC            133   91    5B    [
034   28    1C    FS             134   92    5C    \   '\\'
035   29    1D    GS             135   93    5D    ]
036   30    1E    RS             136   94    5E    ^
037   31    1F    US             137   95    5F    _
040   32    20    SPACE          140   96    60    `
041   33    21    !              141   97    61    a
042   34    22    "              142   98    62    b
043   35    23    #              143   99    63    c
044   36    24    $              144   100   64    d
045   37    25    %              145   101   65    e
046   38    26    &              146   102   66    f
047   39    27    '              147   103   67    g
050   40    28    (              150   104   68    h
051   41    29    )              151   105   69    i
052   42    2A    *              152   106   6A    j
053   43    2B    +              153   107   6B    k
054   44    2C    ,              154   108   6C    l
055   45    2D    -              155   109   6D    m
```

056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	|
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

The most common encoding for ASCII uses the code points as 7-bit bytes, so that the encoding of 'A' for example is 65.

This set is actually US ASCII. Due to European desires for accented characters, some punctuation characters are omitted to form a minimal set, ISO 646, while there are "national variants" with suitable European characters. The page by Jukka Korpela has more information for those interested. We shall not need these variants though.

ISO 8859

Octets are now the standard size for bytes. This allows 128 extra code points for extensions to ASCII. A number of different code sets to capture the repertoires of various subsets of European languages are the ISO 8859 series. ISO 8859-1 is also known as Latin-1 and covers many languages in western Europe, while others in this series cover the rest of Europe and even Hebrew, Arabic and Thai. For example, ISO 8859-5 includes the Cyrillic characters of countries such as Russia, while ISO 8859-8 includes the Hebrew alphabet.

The standard encoding for these character sets is to use their code point as an 8-bit value. For example, the character 'Á' in ISO 8859-1 has the code point 193 and is encoded as 193. All of the ISO 8859 series have the bottom 128 values identical to ASCII, so that the ASCII characters are the same in all of these sets.

The HTML specifications used to recommend the ISO 8859-1 character set. HTML 3.2 was the last one to do so, and after that HTML 4.0 recommended Unicode. In 2010 Google made an estimate that of the pages it sees, about 20% were still in ISO 8859 format while 20% were still in ASCII (Unicode nearing 50% of the web).

Unicode

Neither ASCII nor ISO 8859 cover the languages based on hieroglyphs. Chinese is estimated to have about 20,000 separate characters, with about 5,000 in common use. These need more than a byte, and typically two bytes has been used. There have been many of these two-byte character sets: Big5, EUC-TW, GB2312 and GBK/GBX for Chinese, JIS X 0208 for Japanese, and so on. These encodings are generally not mutually compatable.

Unicode is an embracing standard character set intended to cover all major character sets in use. It includes European, Asian, Indian and many more. It is now up to version 5.2 and has over 107,000 characters. The number of code points now exceeds 65,536, that is. more than 2^16. This has implications for character encodings.

The first 256 code points correspond to ISO 8859-1, with US ASCII as the first 128. There is thus a backward compatibility with these major character sets, as the code points for ISO 8859-1 and ASCII are exactly the same in Unicode. The same is not true for other character sets: for example, while most of the Big5 characters are also in Unicode, the code points are not the same. The page contains one example of a (large) table mapping from Big5 to Unicode.

To represent Unicode characters in a computer system, an encoding must be used. The encoding UCS is a two-byte encoding using the code point values of the Unicode characters. However, since there are now too many characters in Unicode to fit them all into 2 bytes, this encoding is obsolete and no longer used. Instead there are:

- UTF-32 is a 4-byte encoding, but is not commonly used, and HTML 5 warns explicitly against using it
- UTF-16 encodes the most common characters into 2 bytes with a further 2 bytes for the "overflow", with ASCII and ISO 8859-1 having the usual values
- UTF-8 uses between 1 and 4 bytes per character, with ASCII having the usual values (but not ISO 8859-1)
- UTF-7 is used sometimes, but is not common

UTF-8 Go and runes

UTF-8 is the most commonly used encoding. Google estimates that 50% of the pages that it sees are encoded in UTF-8. The ASCII set has the same encoding values in UTF-8, so a UTF-8 reader can read text consisting of just ASCII characters as well as text from the full Unicode set.

Go uses UTF-8 encoded characters in its strings. Each character is of type `rune` . This is a alias for `int32` as a Unicode character can be 1, 2 or 4 bytes in UTF-8 encoding. In terms of characters, a string is an array of `rune` s.

A string is also an array of bytes, but you have to be careful: only for the ASCII subset is a byte equal to a character. All other characters occupy two, three or four bytes. This means that the length of a string in characters (runes) is generally not the same as the length of its byte array. They are only equal when the string consists of ASCII characters only.

The following program fragment illustrates this. If we take a UTF-8 string and test its length, you get the length of the underlying byte array. But if you cast the string to an array of runes `[]rune` then you get an array of the Unicode code points which is generally the number of characters:

```
str := "百度一下，你就知道"

println("String length", len([]rune(str)))
println("Byte length", len(str))
```

prints

```
String length 9
Byte length 27
```

UTF-8 client and server

Possibly surprisingly, you need do nothing special to handle UTF-8 text in either the client or the server. The underlying data type for a UTF-8 string in Go is a byte array, and as we saw just above, Go looks after encoding the string into 1, 2, 3 or 4 bytes as needed. The length of the string is the length of the byte array, so you write any UTF-8 string by writing the byte array.

Similarly to read a string, you just read into a byte array and then cast the array to a string using `string([]byte)` . If Go cannot properly decode bytes into Unicode characters, then it gives the Unicode Replacement Character `\uFFFD` . The length of the resulting byte array is the length of the legal portion of the string.

So the clients and servers given in earlier chapters work perfectly well with UTF-8 encoded text.

ASCII client and server

The ASCII characters have the same encoding in ASCII and in UTF-8. So ordinary UTF-8 character handling works fine for ASCII characters. No special handling need to be done.

UTF-16 and Go

UTF-16 deals with arrays of short 16-bit unsigned integers. The package `utf16` is designed to manage such arrays. To convert a normal Go string, that is a UTF-8 string, into UTF-16, you first extract the code points by coercing it into a `[]rune` and then use `utf16.Encode` to produce an array of type `uint16`.

Similarly, to decode an array of unsigned short UTF-16 values into a Go string, you use `utf16.Decode` to convert it into code points as type `[]rune` and then to a string. The following code fragment illustrates this

```
str := "百度一下，你就知道"

runes := utf16.Encode([]rune(str))
ints := utf16.Decode(runes)

str = string(ints)
```

These type conversions need to be applied by clients or servers as appropriate, to read and write 16-bit short integers, as shown below.

Little-endian and big-endian

Unfortunately, there is a little devil lurking behind UTF-16. It is basically an encoding of characters into 16-bit short integers. The big question is: for each short, how is it written as two bytes? The top one first, or the top one second? Either way is fine, as long as the receiver uses the same convention as the sender.

Unicode has addressed this with a special character known as the BOM (byte order marker). This is a zero-width non-printing character, so you never see it in text. But its value 0xfffe is chosen so that you can tell the byte-order:

- In a big-endian system it is FF FE
- In a little-endian system it is FE FF

Text will sometimes place the BOM as the first character in the text. The reader can then examine these two bytes to determine what endian-ness has been used.

UTF-16 client and server

Using the BOM convention, we can write a server that prepends a BOM and writes a string in UTF-16 as

```
/* UTF16 Server
 */
package main

import (
    "fmt"
    "net"
    "os"
```

```go
    "unicode/utf16"
)

const BOM = '\ufffe'

func main() {

    service := "0.0.0.0:1210"
    tcpAddr, err := net.ResolveTCPAddr("tcp", service)
    checkError(err)

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

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

        str := "j'ai arrêté"
        shorts := utf16.Encode([]rune(str))
        writeShorts(conn, shorts)

        conn.Close() // we're finished
    }
}

func writeShorts(conn net.Conn, shorts []uint16) {
    var bytes [2]byte

    // send the BOM as first two bytes
    bytes[0] = BOM >> 8
    bytes[1] = BOM & 255
    _, err := conn.Write(bytes[0:])
    if err != nil {
        return
    }

    for _, v := range shorts {
        bytes[0] = byte(v >> 8)
        bytes[1] = byte(v & 255)

        _, err = conn.Write(bytes[0:])
        if err != nil {
            return
        }
    }
}

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

while a client that reads a byte stream, extracts and examines the BOM and then decodes the rest of the stream is

```
/* UTF16 Client
 */
package main

import (
    "fmt"
    "net"
    "os"
    "unicode/utf16"
)

const BOM = '\ufffe'

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

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

    shorts := readShorts(conn)
    ints := utf16.Decode(shorts)
    str := string(ints)

    fmt.Println(str)

    os.Exit(0)
}

func readShorts(conn net.Conn) []uint16 {
    var buf [512]byte

    // read everything into the buffer
    n, err := conn.Read(buf[0:2])
    for true {
        m, err := conn.Read(buf[n:])
        if m == 0 || err != nil {
            break
        }
        n += m
    }

    checkError(err)
    var shorts []uint16
    shorts = make([]uint16, n/2)

    if buf[0] == 0xff && buf[1] == 0xfe {
        // big endian
        for i := 2; i < n; i += 2 {
            shorts[i/2] = uint16(buf[i])<<8 + uint16(buf[i+1])
        }
    } else if buf[1] == 0xfe && buf[0] == 0xff {
        // little endian
        for i := 2; i < n; i += 2 {
            shorts[i/2] = uint16(buf[i+1])<<8 + uint16(buf[i])
        }
    } else {
        // unknown byte order
        fmt.Println("Unknown order")
    }
    return shorts

}

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

Unicode gotcha's

This book is not about i18n issues. In particular we don't want to delve into the arcane areas of Unicode. But you should know that Unicode is not a simple encoding and there are many complexities. For example, some earlier character sets used *non-spacing* characters, particularly for accents. This was brought into Unicode, so you can produce accented characters in two ways: as a single Unicode character, or as a pair of non-spacing accent plus non-accented character. For example, U+04D6 CYRILLIC CAPITAL LETTER IE WITH BREVE is a single character. It is equivalent to U+0415 CYRILLIC CAPITAL LETTER IE combined with the breve accent U+0306 COMBINING BREVE. This makes string comparison difficult on occasions. The Go specification does not at present address such issues.

ISO 8859 and Go

The ISO 8859 series are 8-bit character sets for different parts of Europe and some other areas. They all have the ASCII set common in the low part, but differ in the top part. According to Google, ISO 8859 codes account for about 20% of the web pages it sees.

The first code, ISO 8859-1 or Latin-1, has the first 256 characters in common with Unicode. The encoded value of the Latin-1 characters is the same in UTF-16 and in the default ISO 8859-1 encoding. But this doesn't really help much, as UTF-16 is a 16-bit encoding and ISO 8859-1 is an 8-bit encoding. UTF-8 is a 8-bit encoding, but it uses the top bit to signal extra bytes, so only the ASCII subset overlaps for UTF-8 and ISO 8859-1. So UTF-8 doesn't help much either.

But the ISO 8859 series don't have any complex issues. To each character in each set corresponds a unique Unicode character. For example, in ISO 8859-2, the character "latin capital letter I with ogonek" has ISO 8859-2 code point 0xc7 (in hexadecimal) and corresponding Unicode code point of U+012E. Transforming either way between an ISO 8859 set and the corresponding Unicode characters is essentially just a table lookup.

The table from ISO 8859 code points to Unicode code points could be done as an array of 256 integers. But many of these will have the same value as the index. So we just use a map of the different ones, and those not in the map take the index value.

For ISO 8859-2 a portion of the map is

```
var unicodeToISOMap = map[int] uint8 {
    0x12e: 0xc7,
    0x10c: 0xc8,
    0x118: 0xca,
    // plus more
}
```

and a function to convert UTF-8 strings to an array of ISO 8859-2 bytes is

```
/* Turn a UTF-8 string into an ISO 8859 encoded byte array
*/
func unicodeStrToISO(str string) []byte {
        // get the unicode code points
    codePoints := []int(str)

        // create a byte array of the same length
    bytes := make([]byte, len(codePoints))

    for n, v := range(codePoints) {
                // see if the point is in the exception map
        iso, ok := unicodeToISOMap[v]
        if !ok {
                        // just use the value
            iso = uint8(v)
        }
        bytes[n] = iso
    }
    return bytes
}
```

In a similar way you can change an array of ISO 8859-2 bytes into a UTF-8 string:

```
var isoToUnicodeMap = map[uint8] int {
    0xc7: 0x12e,
    0xc8: 0x10c,
    0xca: 0x118,
    // and more
}

func isoBytesToUnicode(bytes []byte) string {
    codePoints := make([]int, len(bytes))
    for n, v := range(bytes) {
        unicode, ok :=isoToUnicodeMap[v]
        if !ok {
            unicode = int(v)
        }
        codePoints[n] = unicode
    }
    return string(codePoints)
}
```

These functions can be used to read and write UTF-8 strings as ISO 8859-2 bytes. By changing the mapping table, you can cover the other ISO 8859 codes. Latin-1, or ISO 8859-1, is a special case - the exception map is empty as the code points for Latin-1 are the same in Unicode. You could also use the same technique for other character sets based on a table mapping, such as Windows 1252.

Other character sets and Go

There are very, very many character set encodings. According to Google, these generally only have a small use, which will hopefully decrease even further in time. But if your software wants to capture all markets, then you may need to handle them.

In the simplest cases, a lookup table will suffice. But that doesn't always work. The character coding ISO 2022 minimised character set sizes by using a finite state machine to swap code pages in and out. This was borrowed by some of the Japanese encodings, and makes things very complex.

Go does not at present give any language or package support for these other character sets. So you either avoid their use, fail to talk to applications that do use them, or write lots of your own code!

Conclusion

There hasn't been much code in this chapter. Instead, there have been some of the concepts of a very complex area. It's up to you: if you want to assume everyone speaks US English then the world is simple. But if you want your applications to be usable by the rest of the world, then you need to pay attention to these complexities.

Chapter 7 Security

Introduction

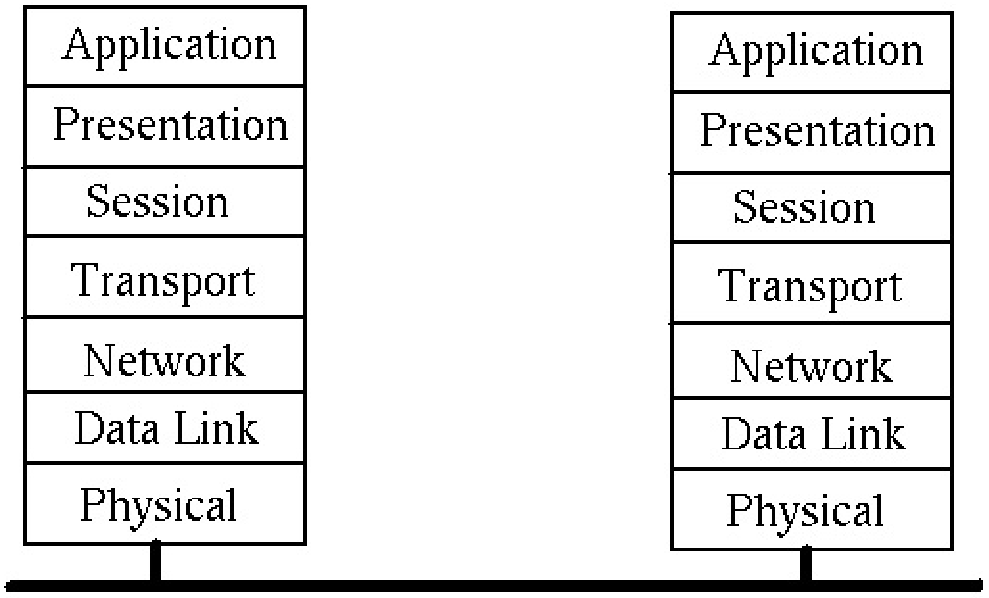
Although the internet was originally designed as a system to withstand attacks by hostile agents, it developed in a co-operative environment of relatively trusted entities. Alas, those days are long gone. Spam mail, denial of service attacks, phishing attempts and so on are indicative that anyone using the internet does so at their own risk.

Applications have to be built to work correctly in hostile situations. "correctly" no longer means just getting the functional aspects of the program correct, but also means ensuring privacy and integrity of data transferred, access only to legitimate users and other issues.

This of course makes your programs much more complex. There are *difficult* and *subtle* computing problems involved in making applications secure. Attempts to do it yourself (such as making up your own encryption libraries) are usually doomed to failure. Instead, you need to make use of libraries designed by security professionals.

ISO security architecture

The ISO OSI (open systems interconnect) seven-layer model of distributed systems is well known and is repeated in this figure:



What is less well known is that ISO built a whole series of documents upon this architecture. For our purposes here, the most important is the ISO Security Architecture model, ISO 7498-2.

Functions and levels

The principal functions required of a security system are

- Authentication - proof of identity
- Data integrity - data is not tampered with
- Confidentiality - data is not exposed to others
- Notarization/signature
- Access control
- Assurance/availability

These are required at the following levels of the OSI stack:

- Peer entity authentication (3, 4, 7)
- Data origin authentication (3, 4, 7)
- Access control service (3, 4, 7)
- Connection confidentiality (1, 2, 3, 4, 6, 7)
- Connectionless confidentiality (1, 2, 3, 4, 6, 7)
- Selective field confidentiality (6, 7)
- Traffic flow confidentiality (1, 3, 7)
- Connection integrity with recovery (4, 7)

- Connection integrity without recovery (4, 7)
- Connection integrity selective field (7)
- Connectionless integrity selective field (7)
- Non-repudiation at origin (7)
- Non-repudiation of receipt (7)

Mechanisms

- Peer entity authentication
  - encryption
  - digital signature
  - authentication exchange
- Data origin authentication
  - encryption
  - digital signature
- Access control service
  - access control lists
  - passwords
  - capabilities lists
  - labels
- Connection confidentiality
  - encryption
  - routing control
- Connectionless confidentiality
  - encryption
  - routing control
- Selective field confidentiality
  - encryption
- Traffic flow confidentiality
  - encryption
  - traffic padding
  - routing control
- Connection integrity with recovery
  - encryption
  - data integrity
- Connection integrity without recovery
  - encryption
  - data integrity
- Connection integrity selective field
  - encryption
  - data integrity
- Connectionless integrity
  - encryption
  - digital signature

    - data integrity
- Connectionless integrity selective field
    - encryption
    - digital signature
    - data integrity
- Non-repudiation at origin
    - digital signature
    - data integrity
    - notarization
- Non-repudiation of receipt
    - digital signature
    - data integrity
    - notarization

Data integrity

Ensuring data integrity means supplying a means of testing that the data has not been tampered with. Usually this is done by forming a simple number out of the bytes in the data. This process is called *hashing* and the resulting number is called a *hash* or *hash value*.

A naive hashing algorithm is just to sum up all the bytes in the data. However, this still allows almost any amount of changing the data around and still preserving the hash values. For example, an attacker could just swap two bytes. This preserves the hash value, but could end up with you owing someone $65,536 instead of $256.

Hashing algorithms used for security purposes have to be "strong", so that it is very difficult for an attacker to find a different sequence of bytes with the same hash value. This makes it hard to modify the data to the attacker's purposes. Security researchers are constantly testing hash algorithms to see if they can break them - that is, find a simple way of coming up with byte sequences to match a hash value. They have devised a series of *cryptographic* hashing algorithms which are believed to be strong.

Go has support for several hashing algorithms, including MD4, MD5, RIPEMD-160, SHA1, SHA224, SHA256, SHA384 and SHA512. They all follow the same pattern as far as the Go programmer is concerned: a function `New` (or similar) in the appropriate package returns a `Hash` object from the `hash` package.

A `Hash` has an `io.Writer` , and you write the data to be hashed to this writer. You can query the number of bytes in the hash value by `Size` and the hash value by `Sum` .

A typical case is MD5 hashing. This uses the `md5` package. The hash value is a 16 byte array. This is typically printed out in ASCII form as four hexadecimal numbers, each made of 4 bytes. A simple program is

```
/* MD5Hash
 */

package main

import (
    "crypto/md5"
    "fmt"
)

func main() {
    hash := md5.New()
    bytes := []byte("hello\n")
    hash.Write(bytes)
    hashValue := hash.Sum(nil)
    hashSize := hash.Size()
    for n := 0; n < hashSize; n += 4 {
        var val uint32
        val = uint32(hashValue[n])<<24 +
            uint32(hashValue[n+1])<<16 +
            uint32(hashValue[n+2])<<8 +
            uint32(hashValue[n+3])
        fmt.Printf("%x ", val)
    }
    fmt.Println()
}
```

which prints `b1946ac9 2492d234 7c6235b4 d2611184`

A variation on this is the HMAC (Keyed-Hash Message Authentication Code) which adds a key to the hash algorithm. There is little change in using this. To use MD5 hashing along with a key, replace the call to `New` by

```
func NewMD5(key []byte) hash.Hash
```

Symmetric key encryption

There are two major mechanisms used for encrypting data. The first uses a single key that is the same for both encryption and decryption. This key needs to be known to both the encrypting and the decrypting agents. How this key is transmitted between the agents is not discussed.

As with hashing, there are many encryption algorithms. Many are now known to have weaknesses, and in general algorithms become weaker over time as computers get faster. Go has support for several symmetric key algorithms such as Blowfish and DES.

The algorithms are *block* algorithms. That is they work on blocks of data. If your data is not aligned to the block size, then you will have to pad it with extra blanks at the end.

Each algorithm is represented by a *Cipher* object. This is created by *NewCipher* in the appropriate package, and takes the symmetric key as parameter.

Once you have a cipher, you can use it to encrypt and decrypt blocks of data. The blocks have to be 8-byte blocks for Blowfish. A program to illustrate this is

```
/* Blowfish
 */

package main

import (
    "bytes"
    "golang.org/x/crypto/blowfish"
    "fmt"
)

func main() {
    key := []byte("my key")
    cipher, err := blowfish.NewCipher(key)
    if err != nil {
        fmt.Println(err.Error())
    }
    src := []byte("hello\n\n\n")
    var enc [512]byte

    cipher.Encrypt(enc[0:], src)

    var decrypt [8]byte
    cipher.Decrypt(decrypt[0:], enc[0:])
    result := bytes.NewBuffer(nil)
    result.Write(decrypt[0:8])
    fmt.Println(string(result.Bytes()))
}
```

Blowfish is not in the Go 1 distribution. You have to install it by running `go get golang.org/x/crypto/blowfish` in a directory where you have source that needs to use it.

Public key encryption

Public key encryption and decryption requires *two* keys: one to encrypt and a second one to decrypt. The encryption key is usually made public in some way so that anyone can encrypt messages to you. The decryption key must stay private, otherwise everyone would be able to decrypt those messages! Public key systems are asymmetric, with different keys for different uses.

There are many public key encryption systems supported by Go. A typical one is the RSA scheme.

A program generating RSA private and public keys is

```
/* GenRSAKeys
 */

package main

import (
    "crypto/rand"
    "crypto/rsa"
    "crypto/x509"
    "encoding/gob"
    "encoding/pem"
    "fmt"
    "os"
)

func main() {
    reader := rand.Reader
    bitSize := 512
    key, err := rsa.GenerateKey(reader, bitSize)
    checkError(err)

    fmt.Println("Private key primes", key.Primes[0].String(), key.Primes[1].String())
    fmt.Println("Private key exponent", key.D.String())

    publicKey := key.PublicKey
    fmt.Println("Public key modulus", publicKey.N.String())
    fmt.Println("Public key exponent", publicKey.E)

    saveGobKey("private.key", key)
    saveGobKey("public.key", publicKey)

    savePEMKey("private.pem", key)
}

func saveGobKey(fileName string, key interface{}) {
    outFile, err := os.Create(fileName)
    checkError(err)
    encoder := gob.NewEncoder(outFile)
    err = encoder.Encode(key)
    checkError(err)
    outFile.Close()
}

func savePEMKey(fileName string, key *rsa.PrivateKey) {

    outFile, err := os.Create(fileName)
    checkError(err)

    var privateKey = &pem.Block{Type: "RSA PRIVATE KEY",
        Bytes: x509.MarshalPKCS1PrivateKey(key)}

    pem.Encode(outFile, privateKey)

    outFile.Close()
}

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

The program also saves the certificates using gob serialisation. They can be read back by this program:

```
/* LoadRSAKeys
 */

package main

import (
    "crypto/rsa"
    "encoding/gob"
    "fmt"
    "os"
)

func main() {
    var key rsa.PrivateKey
    loadKey("private.key", &key)

    fmt.Println("Private key primes", key.Primes[0].String(), key.Primes[1].String())
    fmt.Println("Private key exponent", key.D.String())

    var publicKey rsa.PublicKey
    loadKey("public.key", &publicKey)

    fmt.Println("Public key modulus", publicKey.N.String())
    fmt.Println("Public key exponent", publicKey.E)
}

func loadKey(fileName string, key interface{}) {
    inFile, err := os.Open(fileName)
    checkError(err)
    decoder := gob.NewDecoder(inFile)
    err = decoder.Decode(key)
    checkError(err)
    inFile.Close()
}

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

X.509 certificates

A Public Key Infrastructure (PKI) is a framework for a collection of public keys, along with additional information such as owner name and location, and links between them giving some sort of approval mechanism.

The principal PKI in use today is based on X.509 certificates. For example, web browsers use them to verify the identity of web sites.

An example program to generate a self-signed X.509 certificate for my web site and store it in a `.cer` file is

```
/* GenX509Cert
 */

package main

import (
    "crypto/rand"
    "crypto/rsa"
    "crypto/x509"
    "crypto/x509/pkix"
    "encoding/gob"
    "encoding/pem"
    "fmt"
    "math/big"
    "os"
    "time"
)

func main() {
    random := rand.Reader

    var key rsa.PrivateKey
    loadKey("private.key", &key)

    now := time.Now()
    then := now.Add(60 * 60 * 24 * 365 * 1000 * 1000 * 1000) // one year
    template := x509.Certificate{
        SerialNumber: big.NewInt(1),
        Subject: pkix.Name{
            CommonName:   "jan.newmarch.name",
            Organization: []string{"Jan Newmarch"},
        },
        //    NotBefore: time.Unix(now, 0).UTC(),
        //    NotAfter:  time.Unix(now+60*60*24*365, 0).UTC(),
        NotBefore: now,
        NotAfter:  then,

        SubjectKeyId: []byte{1, 2, 3, 4},
        KeyUsage:     x509.KeyUsageCertSign | x509.KeyUsageKeyEncipherment | x509.KeyUsage
DigitalSignature,
        BasicConstraintsValid: true,
        IsCA:                  true,
        DNSNames:              []string{"jan.newmarch.name", "localhost"},
    }
    derBytes, err := x509.CreateCertificate(random, &template,
        &template, &key.PublicKey, &key)
    checkError(err)

    certCerFile, err := os.Create("jan.newmarch.name.cer")
    checkError(err)
    certCerFile.Write(derBytes)
    certCerFile.Close()

    certPEMFile, err := os.Create("jan.newmarch.name.pem")
    checkError(err)
    pem.Encode(certPEMFile, &pem.Block{Type: "CERTIFICATE", Bytes: derBytes})
    certPEMFile.Close()

    keyPEMFile, err := os.Create("private.pem")
    checkError(err)
    pem.Encode(keyPEMFile, &pem.Block{Type: "RSA PRIVATE KEY",
        Bytes: x509.MarshalPKCS1PrivateKey(&key)})
    keyPEMFile.Close()
}

func loadKey(fileName string, key interface{}) {
    inFile, err := os.Open(fileName)
    checkError(err)
    decoder := gob.NewDecoder(inFile)
    err = decoder.Decode(key)
    checkError(err)
    inFile.Close()
}

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

This can then be read back in by

```
/* ReadX509Cert
 */

package main

import (
    "crypto/x509"
    "fmt"
    "os"
)

func main() {
    certCerFile, err := os.Open("jan.newmarch.name.cer")
    checkError(err)
    derBytes := make([]byte, 1000) // bigger than the file
    count, err := certCerFile.Read(derBytes)
    checkError(err)
    certCerFile.Close()

    // trim the bytes to actual length in call
    cert, err := x509.ParseCertificate(derBytes[0:count])
    checkError(err)

    fmt.Printf("Name %s\n", cert.Subject.CommonName)
    fmt.Printf("Not before %s\n", cert.NotBefore.String())
    fmt.Printf("Not after %s\n", cert.NotAfter.String())

}

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

TLS

Encryption/decryption schemes are of limited use if you have to do all the heavy lifting yourself. The most popular mechanism on the internet to give support for encrypted message passing is currently TLS (Transport Layer Security) which was formerly SSL (Secure Sockets Layer).

In TLS, a client and a server negotiate identity using X.509 certificates. Once this is complete, a secret key is invented between them, and all encryption/decryption is done using this key. The negotiation is relatively slow, but once complete a faster private key mechanism is used.

A server is

```
/* TLSEchoServer
 */
package main

import (
    "crypto/rand"
    "crypto/tls"
    "fmt"
    "net"
    "os"
    "time"
)

func main() {

    cert, err := tls.LoadX509KeyPair("jan.newmarch.name.pem", "private.pem")
    checkError(err)
    config := tls.Config{Certificates: []tls.Certificate{cert}}

    now := time.Now()
    config.Time = func() time.Time { return now }
    config.Rand = rand.Reader

    service := "0.0.0.0:1200"

    listener, err := tls.Listen("tcp", service, &config)
    checkError(err)
    fmt.Println("Listening")
    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println(err.Error())
            continue
        }
        fmt.Println("Accepted")
        go handleClient(conn)
    }
}

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

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

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

The server works with the following client:

```
/* TLSEchoClient
 */
package main

import (
    "fmt"
    "os"
    "crypto/tls"
)

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

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

    for n := 0; n < 10; n++ {
        fmt.Println("Writing...")
        conn.Write([]byte("Hello " + string(n+48)))

        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.Println("Fatal error ", err.Error())
        os.Exit(1)
    }
}
```

Conclusion

Security is a huge area in itself, and in this chapter we have barely touched on it. However, the major concepts have been covered. What has not been stressed is how much security needs to be built into the design phase: security as an afterthought is nearly always a failure.

Chapter 8 HTTP

Introduction

The World Wide Web is a major distributed system, with millions of users. A site may become a Web host by running an HTTP server. While Web clients are typically users with a browser, there are many other "user agents" such as web spiders, web application clients and so on.

The Web is built on top of the HTTP (Hyper-Text Transport Protocol) which is layered on top of TCP. HTTP has been through three publicly available versions, but the latest - version 1.1 - is now the most commonly used.

In this chapter we give an overview of HTTP, followed by the Go APIs to manage HTTP connections.

Overview of HTTP

URLs and resources

URLs specify the location of a *resource*. A resource is often a static file, such as an HTML document, an image, or a sound file. But increasingly, it may be a dynamically generated object, perhaps based on information stored in a database.

When a user agent requests a resource, what is returned is not the resource itself, but some *representation* of that resource. For example, if the resource is a static file, then what is sent to the user agent is a copy of the file.

Multiple URLs may point to the same resource, and an HTTP server will return appropriate representations of the resource for each URL. For example, an company might make product information available both internally and externally using different URLs for the same product. The internal representation of the product might include information such as internal contact officers for the product, while the external representation might include the location of stores selling the product.

This view of resources means that the HTTP protocol can be fairly simple and straightforward, while an HTTP server can be arbitrarily complex. HTTP has to deliver requests from user agents to servers and return a byte stream, while a server might have to do any amount of processing of the request.

HTTP characteristics

HTTP is a stateless, connectionless, reliable protocol. In the simplest form, each request from a user agent is handled reliably and then the connection is broken. Each request involves a separate TCP connection, so if many resources are required (such as images embedded in an HTML page) then many TCP connections have to be set up and torn down in a short space of time.

There are many optimisations in HTTP which add complexity to the simple structure, in order to create a more efficient and reliable protocol.

Versions

There are 3 versions of HTTP

- Version 0.9 - totally obsolete
- Version 1.0 - almost obsolete
- Version 1.1 - current

Each version must understand requests and responses of earlier versions.

HTTP 0.9

Request format

```
Request = Simple-Request

Simple-Request = "GET" SP Request-URI CRLF
```

Response format

A response is of the form

```
Response = Simple-Response

Simple-Response = [Entity-Body]
```

HTTP 1.0

This version added much more information to the requests and responses. Rather than "grow" the 0.9 format, it was just left alongside the new version.

Request format

The format of requests from client to server is

```
Request = Simple-Request | Full-Request

Simple-Request = "GET" SP Request-URI CRLF

Full-Request = Request-Line
        *(General-Header
        | Request-Header
        | Entity-Header)
        CRLF
        [Entity-Body]
```

A Simple-Request is an HTTP/0.9 request and must be replied to by a Simple-Response.

A Request-Line has format

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

where

```
Method = "GET" | "HEAD" | POST |
     extension-method
```

e.g. `GET http://jan.newmarch.name/index.html HTTP/1.0`

Response format

A response is of the form

```
Response = Simple-Response | Full-Response

Simple-Response = [Entity-Body]

Full-Response = Status-Line
        *(General-Header
        | Response-Header
        | Entity-Header)
        CRLF
        [Entity-Body]
```

The Status-Line gives information about the fate of the request:

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

e.g.

```
HTTP/1.0 200 OK

The codes are

Status-Code =      "200" ; OK
        | "201" ; Created
        | "202" ; Accepted
        | "204" ; No Content
        | "301" ; Moved permanently
        | "302" ; Moved temporarily
        | "304" ; Not modified
        | "400" ; Bad request
        | "401" ; Unauthorised
        | "403" ; Forbidden
        | "404" ; Not found
        | "500" ; Internal server error
        | "501" ; Not implemented
        | "502" ; Bad gateway
        | "503" | Service unavailable
        | extension-code
```

The Entity-Header contains useful information about the Entity-Body to follow

```
Entity-Header =    Allow
        | Content-Encoding
        | Content-Length
        | Content-Type
        | Expires
        | Last-Modified
        | extension-header
```

For example

```
HTTP/1.1 200 OK
Date: Fri, 29 Aug 2003 00:59:56 GMT
Server: Apache/2.0.40 (Unix)
Accept-Ranges: bytes
Content-Length: 1595
Connection: close
Content-Type: text/html; charset=ISO-8859-1
```

HTTP 1.1

HTTP 1.1 fixes many problems with HTTP 1.0, but is more complex because of it. This version is done by extending or refining the options available to HTTP 1.0. e.g.

- there are more commands such as TRACE and CONNECT
- you should use absolute URLs, particularly for connecting by proxies e.g

```
GET http://www.w3.org/index.html HTTP/1.1
```

- there are more attributes such as If-Modified-Since, also for use by proxies

The changes include

- hostname identification (allows virtual hosts)
- content negotiation (multiple languages)
- persistent connections (reduces TCP overheads - this is very messy)
- chunked transfers
- byte ranges (request parts of documents)
- proxy support

The 0.9 protocol took one page. The 1.0 protocol was described in about 20 pages. 1.1 takes 120 pages.

Simple user-agents

User agents such as browsers make requests and get responses. The response type is

```
type Response struct {
    Status     string // e.g. "200 OK"
    StatusCode int    // e.g. 200
    Proto      string // e.g. "HTTP/1.0"
    ProtoMajor int    // e.g. 1
    ProtoMinor int    // e.g. 0

    RequestMethod string // e.g. "HEAD", "CONNECT", "GET", etc.

    Header map[string]string

    Body io.ReadCloser

    ContentLength int64

    TransferEncoding []string

    Close bool

    Trailer map[string]string
}
```

We shall examine this data structure through examples. The simplest request is from a user agent is "HEAD" which asks for information about a resource and its HTTP server. The function

```
func Head(url string) (r *Response, err os.Error)
```

can be used to make this query.

The status of the response is in the response field `Status`, while the field `Header` is a map of the header fields in the HTTP response. A program to make this request and display the results is

```
/* Head
 */

package main

import (
    "fmt"
    "net/http"
    "os"
)

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

    response, err := http.Head(url)
    if err != nil {
        fmt.Println(err.Error())
        os.Exit(2)
    }

    fmt.Println(response.Status)
    for k, v := range response.Header {
        fmt.Println(k+":", v)
    }

    os.Exit(0)
}
```

When run against a resource as in `Head http://www.golang.com/` it prints something like

```
200 OK
Content-Type: text/html; charset=utf-8
Date: Tue, 14 Sep 2015 05:34:29 GMT
Cache-Control: public, max-age=3600
Expires: Tue, 14 Sep 2015 06:34:29 GMT
Server: Google Frontend
```

Usually, we are want to retrieve a resource rather than just get information about it. The "GET" request will do this, and this can be done using

```
func Get(url string) (r *Response, finalURL string, err os.Error)
```

The content of the response is in the response field `Body` which is of type `io.ReadCloser`. We can print the content to the screen with the following program

```
/* Get
 */

package main

import (
    "fmt"
    "net/http"
    "net/http/httputil"
    "os"
    "strings"
)

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

    response, err := http.Get(url)
    if err != nil {
        fmt.Println(err.Error())
        os.Exit(2)
    }

    if response.Status != "200 OK" {
        fmt.Println(response.Status)
        os.Exit(2)
    }

    b, _ := httputil.DumpResponse(response, false)
    fmt.Print(string(b))

    contentTypes := response.Header["Content-Type"]
    if !acceptableCharset(contentTypes) {
        fmt.Println("Cannot handle", contentTypes)
        os.Exit(4)
    }

    var buf [512]byte
    reader := response.Body
    for {
        n, err := reader.Read(buf[0:])
        if err != nil {
            os.Exit(0)
        }
        fmt.Print(string(buf[0:n]))
    }
    os.Exit(0)
}

func acceptableCharset(contentTypes []string) bool {
    // each type is like [text/html; charset=UTF-8]
    // we want the UTF-8 only
    for _, cType := range contentTypes {
        if strings.Index(strings.ToUpper(cType), "UTF-8") != -1 {
            return true
        }
    }
    return false
}
```

Note that there are important character set issues of the type discussed in the previous chapter. The server will deliver the content using some character set encoding, and possibly some transfer encoding. Usually this is a matter of negotiation between user agent and server, but the simple `Get` command that we are using does not include the user agent component of the negotiation. So the server can send whatever character encoding it wishes.

At the time of first writing, I was in China. When I tried this program on `www.google.com`, Google's server tried to be helpful by guessing my location and sending me the text in the Chinese character set Big5! How to tell the server what character encoding is okay for me is discussed later.

Configuring HTTP requests

Go also supplies a lower-level interface for user agents to communicate with HTTP servers. As you might expect, not only does it give you more control over the client requests, but requires you to spend more effort in building the requests. However, there is only a small increase.

The data type used to build requests is the type `Request`. This is a complex type, and is given in the Go documentation as

```
type Request struct {
    Method     string // GET, POST, PUT, etc.
    RawURL     string // The raw URL given in the request.
    URL        *URL   // Parsed URL.
    Proto      string // "HTTP/1.0"
    ProtoMajor int    // 1
    ProtoMinor int    // 0


    // A header maps request lines to their values.
    // If the header says
    //
    //    accept-encoding: gzip, deflate
    //    Accept-Language: en-us
    //    Connection: keep-alive
    //
    // then
    //
    //    Header = map[string]string{
    //        "Accept-Encoding": "gzip, deflate",
    //        "Accept-Language": "en-us",
    //        "Connection": "keep-alive",
    //    }
    //
    // HTTP defines that header names are case-insensitive.
    // The request parser implements this by canonicalizing the
    // name, making the first character and any characters
    // following a hyphen uppercase and the rest lowercase.
    Header map[string]string

    // The message body.
    Body io.ReadCloser

    // ContentLength records the length of the associated content.
    // The value -1 indicates that the length is unknown.
    // Values >= 0 indicate that the given number of bytes may be read from Body.
    ContentLength int64

    // TransferEncoding lists the transfer encodings from outermost to innermost.
    // An empty list denotes the "identity" encoding.
    TransferEncoding []string

    // Whether to close the connection after replying to this request.
    Close bool

    // The host on which the URL is sought.
    // Per RFC 2616, this is either the value of the Host: header
    // or the host name given in the URL itself.
    Host string

    // The referring URL, if sent in the request.
    //
    // Referer is misspelled as in the request itself,
    // a mistake from the earliest days of HTTP.
    // This value can also be fetched from the Header map
    // as Header["Referer"]; the benefit of making it
    // available as a structure field is that the compiler
    // can diagnose programs that use the alternate
    // (correct English) spelling req.Referrer but cannot
    // diagnose programs that use Header["Referrer"].
    Referer string

    // The User-Agent: header string, if sent in the request.
    UserAgent string

    // The parsed form. Only available after ParseForm is called.
    Form map[string][]string

    // Trailer maps trailer keys to values.  Like for Header, if the
    // response has multiple trailer lines with the same key, they will be
    // concatenated, delimited by commas.
    Trailer map[string]string
}
```

There is a lot of information that can be stored in a request. You do not need to fill in all fields, only those of interest. The simplest way to create a request with default values is by for example

```
request, err := http.NewRequest("GET", url.String(), nil)
```

Once a request has been created, you can modify fields. For example, to specify that you only wish to receive UTF-8, add an "Accept-Charset" field to a request by

```
request.Header.Add("Accept-Charset", "UTF-8;q=1, ISO-8859-1;q=0")
```

(Note that the default set ISO-8859-1 always gets a value of one unless mentioned explicitly in the list.).

A client setting a charset request is simple by the above. But there is some confusion about what happens with the server's return value of a charset. The returned resource *should* have a `Content-Type` which will specify the media type of the content such as `text/html`. If appropriate the media type should state the charset, such as `text/html; charset=UTF-8`. If there is no charset specification, then according to the HTTP specification it should be treated as the default ISO8859-1 charset. But the HTML 4 specification states that since many servers don't conform to this, then you can't make any assumptions.

If there is a charset specified in the server's `Content-Type`, then assume it is correct. if there is none specified, since 50% of pages are in UTF-8 and 20% are in ASCII then it is safe to assume UTF-8. Only 30% of pages may be wrong :-(.

The Client object

To send a request to a server and get a reply, the convenience object `Client` is the easiest way. This object can manage multiple requests and will look after issues such as whether the server keeps the TCP connection alive, and so on.

This is illustrated in the following program

```
/* ClientGet
 */

package main

import (
    "fmt"
    "net/http"
    "net/url"
    "os"
    "strings"
)

func main() {
    if len(os.Args) != 2 {
        fmt.Println("Usage: ", os.Args[0], "http://host:port/page")
        os.Exit(1)
    }
    url, err := url.Parse(os.Args[1])
    checkError(err)

    client := &http.Client{}

    request, err := http.NewRequest("GET", url.String(), nil)
    // only accept UTF-8
    request.Header.Add("Accept-Charset", "UTF-8;q=1, ISO-8859-1;q=0")
    checkError(err)

    response, err := client.Do(request)
    if response.Status != "200 OK" {
        fmt.Println(response.Status)
        os.Exit(2)
    }

    chSet := getCharset(response)
    fmt.Printf("got charset %s\n", chSet)
    if chSet != "UTF-8" {
        fmt.Println("Cannot handle", chSet)
        os.Exit(4)
    }

    var buf [512]byte
    reader := response.Body
    fmt.Println("got body")
    for {
        n, err := reader.Read(buf[0:])
        if err != nil {
            os.Exit(0)
        }
        fmt.Print(string(buf[0:n]))
    }

    os.Exit(0)
}

func getCharset(response *http.Response) string {
    contentType := response.Header.Get("Content-Type")
    if contentType == "" {
        // guess
        return "UTF-8"
    }
    idx := strings.Index(contentType, "charset:")
    if idx == -1 {
        // guess
        return "UTF-8"
    }
    return strings.Trim(contentType[idx:], " ")
}

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

Proxy handling

Simple proxy

HTTP 1.1 laid out how HTTP should work through a proxy. A "GET" request should be made to a proxy. However, the URL requested should be the full URL of the destination. In addition the HTTP header should contain a "Host" field, set to the proxy. As long as the proxy is configured to pass such requests through, then that is all that needs to be done.

Go considers this to be part of the HTTP transport layer. To manage this it has a class `Transport`. This contains a field which can be set to a *function* that returns a URL for a proxy. If we have a URL as a string for the proxy, the appropriate transport object is created and then given to a client object by

```
proxyURL, err := url.Parse(proxyString)
transport := &http.Transport{Proxy: http.ProxyURL(proxyURL)}
client := &http.Client{Transport: transport}
```

The client can then continue as before.

The following program illustrates this:

```
/* ProxyGet
 */

package main

import (
    "fmt"
    "io"
    "net/http"
    "net/http/httputil"
    "net/url"
    "os"
)

func main() {
    if len(os.Args) != 3 {
        fmt.Println("Usage: ", os.Args[0], "http://proxy-host:port http://host:port/page")
        os.Exit(1)
    }
    proxyString := os.Args[1]
    proxyURL, err := url.Parse(proxyString)
    checkError(err)
    rawURL := os.Args[2]
    url, err := url.Parse(rawURL)
    checkError(err)

    transport := &http.Transport{Proxy: http.ProxyURL(proxyURL)}
    client := &http.Client{Transport: transport}
    request, err := http.NewRequest("GET", url.String(), nil)

    dump, _ := httputil.DumpRequest(request, false)
    fmt.Println(string(dump))

    response, err := client.Do(request)

    checkError(err)
    fmt.Println("Read ok")

    if response.Status != "200 OK" {
        fmt.Println(response.Status)
        os.Exit(2)
    }
    fmt.Println("Reponse ok")

    var buf [512]byte
    reader := response.Body
    for {
        n, err := reader.Read(buf[0:])
        if err != nil {
            os.Exit(0)
        }
        fmt.Print(string(buf[0:n]))
    }

    os.Exit(0)
}

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

If you have a proxy at, say, XYZ.com on port 8080, test this by

```
go run ProxyGet.go http://XYZ.com:8080/ http://www.google.com
```

If you don't have a suitable proxy to test this, then download and install the Squid proxy to your own computer.

The above program used a known proxy passed as an argument to the program. There are many ways in which proxies can be made known to applications. Most browsers have a configuration menu in which you can enter proxy information: such information is not available to a Go application. Some applications may get proxy information from an `autoproxy.pac` file somewhere in your network: Go does not (yet) know how to parse these JavaScript files and so cannot use them. Linux systems using Gnome have a configuration system called `gconf` in which proxy information can be stored: Go cannot access this. *But* it can find proxy information if it is set in operating system environment variables such as HTTP_PROXY or http_proxy using the function

```
func ProxyFromEnvironment(req *Request) (*url.URL, error)
```

If your programs are running in such an environment you can use this function instead of having to explicitly know the proxy parameters.

Authenticating proxy

Some proxies will require authentication, by a user name and password in order to pass requests. A common scheme is "basic authentication" in which the user name and password are concatenated into a string "user:password" and then BASE64 encoded. This is then given to the proxy by the HTTP request header "Proxy-Authorisation" with the flag that it is the basic authentication

The following program illustrates this, adding the Proxy-Authentication header to the previous proxy program:

```
/* ProxyAuthGet
 */

package main

import (
    "encoding/base64"
    "fmt"
    "io"
    "net/http"
    "net/http/httputil"
    "net/url"
    "os"
)

const auth = "jannewmarch:mypassword"

func main() {
    if len(os.Args) != 3 {
        fmt.Println("Usage: ", os.Args[0], "http://proxy-host:port http://host:port/page")
        os.Exit(1)
    }
    proxy := os.Args[1]
    proxyURL, err := url.Parse(proxy)
    checkError(err)
    rawURL := os.Args[2]
    url, err := url.Parse(rawURL)
    checkError(err)

    // encode the auth
    basic := "Basic " + base64.StdEncoding.EncodeToString([]byte(auth))

    transport := &http.Transport{Proxy: http.ProxyURL(proxyURL)}
    client := &http.Client{Transport: transport}

    request, err := http.NewRequest("GET", url.String(), nil)

    request.Header.Add("Proxy-Authorization", basic)
    dump, _ := httputil.DumpRequest(request, false)
    fmt.Println(string(dump))

    // send the request
    response, err := client.Do(request)

    checkError(err)
    fmt.Println("Read ok")

    if response.Status != "200 OK" {
        fmt.Println(response.Status)
        os.Exit(2)
    }
    fmt.Println("Reponse ok")

    var buf [512]byte
    reader := response.Body
    for {
        n, err := reader.Read(buf[0:])
        if err != nil {
            os.Exit(0)
        }
        fmt.Print(string(buf[0:n]))
    }

    os.Exit(0)
}

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

HTTPS connections by clients

For secure, encrypted connections, HTTP uses TLS which is described in the chapter on security. The protocol of HTTP+TLS is called HTTPS and uses `https://` urls instead of `http://` urls.

Servers are required to return valid X.509 certificates before a client will accept data from them. If the certificate is valid, then Go handles everything under the hood and the clients given previously run okay with https URLs.

Many sites have invalid certificates. They may have expired, they may be self-signed instead of by a recognised Certificate Authority or they may just have errors (such as having an incorrect server name). Browsers such as Firefox put a big warning notice with a "Get me out of here!" button, but you can carry on at your risk - which many people do.

Go presently bails out when it encounters certificate errors. There is cautious support for carrying on but I haven't got it working yet. So there is no current example for "carrying on in the face of adversity :-)". Maybe later.

Servers

The other side to building a client is a Web server handling HTTP requests. The simplest - and earliest - servers just returned copies of files. However, any URL can now trigger an arbitrary computation in current servers.

File server

We start with a basic file server. Go supplies a *multi-plexer*, that is, an object that will read and interpret requests. It hands out requests to *handlers* which run in their own thread. Thus much of the work of reading HTTP requests, decoding them and branching to suitable functions in their own thread is done for us.

For a file server, Go also gives a `FileServer` object which knows how to deliver files from the local file system. It takes a "root" directory which is the top of a file tree in the local system, and a pattern to match URLs against. The simplest pattern is "/" which is the top of any URL. This will match all URLs.

An HTTP server delivering files from the local file system is almost embarrassingly trivial given these objects. It is

```go
/* File Server
 */

package main

import (
    "fmt"
    "net/http"
    "os"
)

func main() {
    // deliver files from the directory /var/www
    //fileServer := http.FileServer(http.Dir("/var/www"))
    fileServer := http.FileServer(http.Dir("/home/httpd/html/"))

    // register the handler and deliver requests to it
    err := http.ListenAndServe(":8000", fileServer)
    checkError(err)
    // That's it!
}

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

This server even delivers `"404 not found"` messages for requests for file resources that don't exist!

Handler functions

In this last program, the handler was given in the second argument to `ListenAndServe`. Any number of handlers can be registered first by calls to `Handle` or `HandleFunc`, with signatures

```
func Handle(pattern string, handler Handler)
func HandleFunc(pattern string, handler func(*Conn, *Request))
```

The second argument to `HandleAndServe` could be `nil`, and then calls are dispatched to all registered handlers. Each handler should have a different URL pattern. For example, the file handler might have URL pattern "/" while a function handler might have URL pattern "/cgi-bin". A more specific pattern takes precedence over a more general pattern.

Common CGI programs are `test-cgi` (written in the shell) or `printenv` (written in Perl) which print the values of the environment variables. A handler can be written to work in a similar manner.

```
/* Print Env
 */

package main

import (
    "fmt"
    "net/http"
    "os"
)

func main() {
    // file handler for most files
    fileServer := http.FileServer(http.Dir("/var/www"))
    http.Handle("/", fileServer)

    // function handler for /cgi-bin/printenv
    http.HandleFunc("/cgi-bin/printenv", printEnv)

    // deliver requests to the handlers
    err := http.ListenAndServe(":8000", nil)
    checkError(err)
    // That's it!
}

func printEnv(writer http.ResponseWriter, req *http.Request) {
    env := os.Environ()
    writer.Write([]byte("<h1>Environment</h1>\n<pre>"))
    for _, v := range env {
        writer.Write([]byte(v + "\n"))
    }
    writer.Write([]byte("</pre>"))
}

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

Note: for simplicity this program does not deliver well-formed HTML. It is missing html, head and body tags.

Using the `cgi-bin` directory in this program is a bit cheeky: it doesn't call an external program like CGI scripts do. It just calls a Go function. Go does have the ability to call external programs using `os.ForkExec`, but does not yet have support for dynamically linkable modules like Apache's `mod_perl`.

Bypassing the default multiplexer

HTTP requests received by a Go server are usually handled by a multiplexer the examines the path in the HTTP request and calls the appropriate file handler, etc. You can define your own handlers. These can either be registered with the default multiplexer by calling `http.HandleFunc` which takes a pattern and a function. The functions such as `ListenAndServe` then take a `nil` handler function. This was done in the last example.

If you want to take over the multiplexer role then you can give a non-zero function as the handler function. This function will then be totally responsible for managing the requests and responses.

The following example is trivial, but illustrates the use of this: the multiplexer function simply returns a `"204 No content"` for all requests:

```
/* ServerHandler
 */

package main

import (
    "net/http"
)

func main() {

    myHandler := http.HandlerFunc(func(rw http.ResponseWriter, request *http.Request) {
        // Just return no content - arbitrary headers can be set, arbitrary body
        rw.WriteHeader(http.StatusNoContent)
    })

    http.ListenAndServe(":8080", myHandler)
}
```

Arbitrarily complex behaviour can be built, of course.

Low-level servers

Go also supplies a lower-level interface for servers. Again, this means that as the programmer you have to do more work. You first make a TCP server, and then wrap a `ServerConn` around it. Then you read `Request`'s and write `Response`'s.

Conclusion

Go has extensive support for HTTP. This is not surprising, since Go was partly invented to fill a need by Google for their own servers.

Chapter 9 Templates

Many languages have mechanisms to convert strings from one to another. Go has a template mechanism to convert strings based on the content of an object supplied as an argument. While this is often used in rewriting HTML to insert object values, it can be used in other situations. Note that this material doesn't have anything explicitly to do with networking, but may be useful to network programs.

Introduction

Most server-side languages have a mechanism for taking predominantly static pages and inserting a dynamically generated component, such as a list of items. Typical examples are scripts in Java Server Pages, PHP scripting and many others. Go has adopted a relatively simple scripting language in the `template` package.

We describe the new package here. The package is designed to take text as input and output different text, based on transforming the original text using the values of an object. Unlike JSP or similar, it is not restricted to HTML files but it is likely to find greatest use there.

The original source is called a *template* and will consist of text that is transmitted unchanged, and embedded commands which can act on and change text. The commands are delimited by `{{ ... }}` , similar to the JSP commands `<%= ... =%>` and PHPs `<?php ... ?>` .

Inserting object values

A template is applied to a Go object. Fields from that Go object can be inserted into the template, and you can 'dig' into the object to find subfields, etc. The current object is represented as `'.'`, so that to insert the value of the current object as a string, you use `{{.}}`. The package uses the `fmt` package by default to work out the string used as inserted values.

To insert the value of a field of the current object, you use the field name prefixed by `'.'`. For example, if the object is of type

```
type Person struct {
        Name      string
        Age       int
        Emails     []string
        Jobs       []*Jobs
}
```

then you insert the values of `Name` and `Age` by

```
The name is {{.Name}}.
The age is {{.Age}}.
```

We can loop over the elements of an array or other list using the `range` command. So to access the contents of the `Emails` array we do

```
{{range .Emails}}
        ...
{{end}}
```

if `Job` is defined by

```
type Job struct {
    Employer string
    Role     string
}
```

and we want to access the fields of a `Person`'s `Jobs`, we can do it as above with a `{{range .Jobs}}`. An alternative is to switch the current object to the `Jobs` field. This is done using the `{{with ...}} ... {{end}}` construction, where now `{{.}}` is the `Jobs` field, which is an array:

```
{{with .Jobs}}
    {{range .}}
        An employer is {{.Employer}}
        and the role is {{.Role}}
    {{end}}
{{end}}
```

You can use this with any field, not just an array. Using templates

Once we have a template, we can apply it to an object to generate a new string, using the object to fill in the template values. This is a two-step process which involves parsing the template and then applying it to an object. The result is output to a `Writer`, as in

```
t := template.New("Person template")
t, err := t.Parse(templ)
if err == nil {
    buff := bytes.NewBufferString("")
    t.Execute(buff, person)
}
```

An example program to apply a template to an object and print to standard output is

```
/**
 * PrintPerson
 */

package main

import (
    "fmt"
    "html/template"
    "os"
)

type Person struct {
    Name   string
    Age    int
    Emails []string
    Jobs   []*Job
}

type Job struct {
    Employer string
    Role     string
}

const templ = `The name is {{.Name}}.
The age is {{.Age}}.
{{range .Emails}}
        An email is {{.}}
{{end}}

{{with .Jobs}}
    {{range .}}
        An employer is {{.Employer}}
        and the role is {{.Role}}
    {{end}}
{{end}}
`
```

```
func main() {
    job1 := Job{Employer: "Monash", Role: "Honorary"}
    job2 := Job{Employer: "Box Hill", Role: "Head of HE"}

    person := Person{
        Name:   "jan",
        Age:    50,
        Emails: []string{"jan@newmarch.name", "jan.newmarch@gmail.com"},
        Jobs:   []*Job{&job1, &job2},
    }

    t := template.New("Person template")
    t, err := t.Parse(templ)
    checkError(err)

    err = t.Execute(os.Stdout, person)
    checkError(err)
}

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

The output from this is

```
The name is jan.
The age is 50.

An email is jan@newmarch.name

An email is jan.newmarch@gmail.com

An employer is Monash
and the role is Honorary

An employer is Box Hill
and the role is Head of HE
```

Note that there is plenty of whitespace as newlines in this printout. This is due to the whitespace we have in our template. If we wish to reduce this, eliminate newlines in the template as in

```
{{range .Emails}} An email is {{.}} {{end}}
```

In the example, we used a string in the program as the template. You can also load templates from a file using the function `template.ParseFiles()` . For some reason that I don't understand (and which wasn't required in earlier versions), the name assigned to the template must be the same as the basename of the first file in the list of files. Is this a bug?

Pipelines

The above transformations insert pieces of text into a template. Those pieces of text are essentially arbitrary, whatever the string values of the fields are. If we want them to appear as part of an HTML document (or other specialised form) then we will have to escape particular sequences of characters. For example, to display arbitrary text in an HTML document we have to change `"<"` to `"&lt;"`. The Go templates have a number of builtin functions, and one of these is the function html. These functions act in a similar manner to Unix pipelines, reading from standard input and writing to standard output.

To take the value of the current object `'.'` and apply HTML escapes to it, you write a "pipeline" in the template

```
{{. | html}}
```

and similarly for other functions.

Mike Samuel has pointed out a convenience function currently in the `exp/template/html` package. If all of the entries in a template need to be passed through the html template function, then the Go function `Escape(t *template.Template)` can take a template and add the `html` function to each node in the template that doesn't already have one. This will be useful for templates used for HTML documents and can form a pattern for similar function uses elsewhere.

Defining functions

The templates use the string representation of an object to insert values, using the `fmt` package to convert the object to a string. Sometimes this isn't what is needed. For example, to avoid spammers getting hold of email addresses it is quite common to see the symbol `'@'` replaced by the word `" at "`, as in `"jan at newmarch.name"`. If we want to use a template to display email addresses in that form, then we have to build a custom function to do this transformation.

Each template function has a name that is used in the templates themselves, and an associated Go function. These are linked by the type

```
type FuncMap map[string]interface{}
```

For example, if we want our template function to be `"emailExpand"` which is linked to the Go function `EmailExpander` then we add this to the functions in a template by

```
t = t.Funcs(template.FuncMap{"emailExpand": EmailExpander})
```

The signature for `EmailExpander` is typically

```
func EmailExpander(args ...interface{}) string
```

In the use we are interested in, there should only be one argument to the function which will be a string. Existing functions in the Go template library have some initial code to handle non-conforming cases, so we just copy that. Then it is just simple string manipulation to change the format of the email address. A program is

```
/**
 * PrintEmails
 */

package main

import (
    "fmt"
    "os"
    "strings"
    "text/template"
)

type Person struct {
    Name   string
    Emails []string
}

const templ = `The name is {{.Name}}.
{{range .Emails}}
        An email is "{{. | emailExpand}}"
```

```
{{end}}
`

func EmailExpander(args ...interface{}) string {
    ok := false
    var s string
    if len(args) == 1 {
        s, ok = args[0].(string)
    }
    if !ok {
        s = fmt.Sprint(args...)
    }

    // find the @ symbol
    substrs := strings.Split(s, "@")
    if len(substrs) != 2 {
        return s
    }
    // replace the @ by " at "
    return (substrs[0] + " at " + substrs[1])
}

func main() {
    person := Person{
        Name:   "jan",
        Emails: []string{"jan@newmarch.name", "jan.newmarch@gmail.com"},
    }

    t := template.New("Person template")

    // add our function
    t = t.Funcs(template.FuncMap{"emailExpand": EmailExpander})

    t, err := t.Parse(templ)

    checkError(err)

    err = t.Execute(os.Stdout, person)
    checkError(err)
}

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

The output is

```
The name is jan.

An email is "jan at newmarch.name"

An email is "jan.newmarch at gmail.com"
```

Variables

The template package allows you to define and use variables. As motivation for this, consider how we might print each person's email address *prefixed* by their name. The type we use is again

```
type Person struct {
        Name      string
        Emails     []string
}
```

To access the email strings, we use a `range` statement such as

```
{{range .Emails}}
    {{.}}
{{end}}
```

But at that point we cannot access the `Name` field as `'.'` is now traversing the array elements and the `Name` is outside of this scope. The solution is to save the value of the `Name` field in a variable that can be accessed anywhere in its scope. Variables in templates are prefixed by `'$'`. So we write

```
{{$name := .Name}}
{{range .Emails}}
    Name is {{$name}}, email is {{.}}
{{end}}
```

The program is

```
/**
 * PrintNameEmails
 */

package main

import (
    "html/template"
    "os"
    "fmt"
)

type Person struct {
    Name   string
    Emails []string
}

const templ = `{{$name := .Name}}
{{range .Emails}}
    Name is {{$name}}, email is {{.}}
{{end}}
```

```
`

func main() {
    person := Person{
        Name:   "jan",
        Emails: []string{"jan@newmarch.name", "jan.newmarch@gmail.com"},
    }

    t := template.New("Person template")
    t, err := t.Parse(templ)
    checkError(err)

    err = t.Execute(os.Stdout, person)
    checkError(err)
}

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

with output

```
Name is jan, email is jan@newmarch.name

Name is jan, email is jan.newmarch@gmail.com
```

Conditional statements

Continuing with our `Person` example, supposing we just want to print out the list of emails, without digging into it. We can do that with a template

```
Name is {{.Name}}
Emails are {{.Emails}}
```

This will print

```
Name is jan
Emails are [jan@newmarch.name jan.newmarch@gmail.com]
```

because that is how the `fmt` package will display a list.

In many circumstances that may be fine, if that is what you want. Let's consider a case where it is *almost* right but not quite. There is a JSON package to serialise objects, which we looked at in Chapter 4. This would produce

```
{"Name": "jan",
 "Emails": ["jan@newmarch.name", "jan.newmarch@gmail.com"]
}
```

The JSON package is the one you would use in practice, but let's see if we can produce JSON output using templates. We can do something similar just by the templates we have. This is *almost* right as a JSON serialiser:

```
{"Name": "{{.Name}}",
 "Emails": {{.Emails}}
}
```

It will produce

```
{"Name": "jan",
 "Emails": [jan@newmarch.name jan.newmarch@gmail.com]
}
```

which has two problems: the addresses aren't in quotes, and the list elements should be `','` separated.

How about this: looking at the array elements, putting them in quotes and adding commas?

```
{"Name": {{.Name}},
  "Emails": [
   {{range .Emails}}
      "{{.}}",
    {{end}}
   ]
 }
```

which will produce

```
{"Name": "jan",
 "Emails": ["jan@newmarch.name", "jan.newmarch@gmail.com",]
}
```

(plus some white space.).

Again, almost correct, but if you look carefully, you will see a trailing `','` after the last list element. According to the JSON syntax (see json.org, this trailing `','` is not allowed. Implementations may vary in how they deal with this.

What we want is "print every element followed by a `','` except for the last one. This is actually a bit hard to do, so a better way is "print every element preceded by a `','` except for the first one. (I got this tip from "brianb" at Stack Overflow.). This is easier, because the first element has index zero and many programming languages, including the Go template language, treat zero as Boolean false.

One form of the conditional statement is `{{if pipeline}} T1 {{else}} T0 {{end}}`. We need the `pipeline` to be the index into the array of emails. Fortunately, a variation on the `range` statement gives us this. There are two forms which introduce variables

```
{{range $elmt := array}}
{{range $index, $elmt := array}}
```

So we set up a loop through the array, and if the index is false (0) we just print the element, otherwise print it preceded by a `','`. The template is

```
{"Name": "{{.Name}}",
 "Emails": [
 {{range $index, $elmt := .Emails}}
    {{if $index}}
        , "{{$elmt}}"
    {{else}}
         "{{$elmt}}"
    {{end}}
 {{end}}
 ]
}
```

and the full program is

```
/**
 * PrintJSONEmails
 */
```

```
package main

import (
    "html/template"
    "os"
    "fmt"
)

type Person struct {
    Name   string
    Emails []string
}

const templ = `{"Name": "{{.Name}}",
 "Emails": [
{{range $index, $elmt := .Emails}}
    {{if $index}}
        , "{{$elmt}}"
    {{else}}
         "{{$elmt}}"
    {{end}}
{{end}}
 ]
}
`

func main() {
    person := Person{
        Name:   "jan",
        Emails: []string{"jan@newmarch.name", "jan.newmarch@gmail.com"},
    }

    t := template.New("Person template")
    t, err := t.Parse(templ)
    checkError(err)

    err = t.Execute(os.Stdout, person)
    checkError(err)
}

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

This gives the correct JSON output.

Before leaving this section, we note that the problem of formatting a list with comma separators can be approached by defining suitable functions in Go that are made available as template functions. To re-use a well known saying, "There's more than one way to do it!". The following program was sent to me by Roger Peppe:

```
/**
 * Sequence.go
 * Copyright Roger Peppe
 */

package main

import (
    "errors"
    "fmt"
    "os"
    "text/template"
)

var tmpl = `{{$comma := sequence "" ", "}}
{{range $}}{{$comma.Next}}{{.}}{{end}}
{{$comma := sequence "" ", "}}
{{$colour := cycle "black" "white" "red"}}
{{range $}}{{$comma.Next}}{{.}} in {{$colour.Next}}{{end}}
`

var fmap = template.FuncMap{
    "sequence": sequenceFunc,
    "cycle":    cycleFunc,
}

func main() {
    t, err := template.New("").Funcs(fmap).Parse(tmpl)
    if err != nil {
        fmt.Printf("parse error: %v\n", err)
        return
    }
    err = t.Execute(os.Stdout, []string{"a", "b", "c", "d", "e", "f"})
    if err != nil {
        fmt.Printf("exec error: %v\n", err)
    }
}

type generator struct {
    ss []string
    i  int
    f  func(s []string, i int) string
}

func (seq *generator) Next() string {
    s := seq.f(seq.ss, seq.i)
    seq.i++
    return s
}

func sequenceGen(ss []string, i int) string {
    if i >= len(ss) {
        return ss[len(ss)-1]
    }
    return ss[i]
}

func cycleGen(ss []string, i int) string {
    return ss[i%len(ss)]
}

func sequenceFunc(ss ...string) (*generator, error) {
    if len(ss) == 0 {
        return nil, errors.New("sequence must have at least one element")
    }
    return &generator{ss, 0, sequenceGen}, nil
}

func cycleFunc(ss ...string) (*generator, error) {
    if len(ss) == 0 {
        return nil, errors.New("cycle must have at least one element")
    }
    return &generator{ss, 0, cycleGen}, nil
}
```

Conclusion

The Go template package is useful for certain kinds of text transformations involving inserting values of objects. It does not have the power of, say, regular expressions, but is faster and in many cases will be easier to use than regular expressions.

Chapter 10 A Complete Web Server

This chapter is principally a lengthy illustration of the HTTP chapter, building a complete Web server in Go. It also shows how to use templates in order to use expressions in text files to insert variable values and to generate repeated sections.

Introduction

I am learning Chinese. Rather, after many years of trying I am still *attempting* to learn Chinese. Of course, rather than buckling down and getting on with it, I have tried all sorts of technical aids. I tried DVDs, videos, flashcards and so on. Eventually I realised that there *wasn't a good computer program for Chinese flashcards*, and so in the interests of learning, I needed to build one.

I had found a program in Python to do some of the task. But sad to say it wasn't well written and after a few attempts at turning it upside down and inside out I came to the conclusion that it was better to start from scratch. Of course, a Web solution would be far better than a standalone one, because then all the other people in my Chinese class could share it, as well as any other learners out there. And of course, the server would be written in Go.

The flashcards server is running at cict.bhtafe.edu.au:8000. The front page consists of a list of flashcard sets currently available, how you want a set displayed (random card order, Chinese, English or random), whether to display a set, add to it, etc. I've spent too much time building it - somehow my Chinese hasn't progressed much while I was doing it... It probably won't be too exciting as a program if you don't want to learn Chinese, but let's get into the structure.

Static pages

Some pages will just have static content. These can be managed by a `fileServer`. For simplicity I put all of the static HTML pages and CSS files in the `html` directory and all of the JavaScript files in the `jscript` directory. These are then delivered by the Go code

```
fileServer := http.FileServer("jscript", "/jscript/")
http.Handle("/jscript/", fileServer)

fileServer = http.FileServer("html", "/html/")
http.Handle("/html/", fileServer)
```

Templates

The list of flashcard sets is open ended, depending on the number of files in a directory. These should not be hard-coded into an HTML page, but the content should be generated as needed. This is an obvious candidate for templates.

The list of files in a directory is generated as a list of strings. These can then be displayed in a table using the template

```
<table>
  {{range .}}
  <tr>
    <td>
      {{.}}
    </td>
  </tr>
</table>
```

The Chinese Dictionary

Chinese is a complex language (aren't they all :-( ). The written form is hieroglyphic, that is "pictograms" instead of using an alphabet. But this written form has evolved over time, and even recently split into two forms: "traditional" Chinese as used in Taiwan and Hong Kong, and "simplified" Chinese as used in mainland China. While most of the characters are the same, about 1,000 are different. Thus a Chinese dictionary will often have two written forms of the same character.

Most Westerners like me can't understand these characters. So there is a "Latinised" form called Pinyin which writes the characters in a phonetic alphabet based on the Latin alphabet. It isn't quite the Latin alphabet, because Chinese is a tonal language, and the Pinyin form has to show the tones (much like acccents in French and other European languages). So a typical dictionary has to show four things: the traditional form, the simplified form, the Pinyin and the English. For example,

Traditional	Simplified	Pinyin	English
好	好	hǎo	good

But again there is a little complication. There is a free Chinese/English dictionary and even better, you can download it as a UTF-8 file, which Go is well suited to handle. In this, the Chinese characters are written in Unicode but the Pinyin characters are not: although there are Unicode characters for letters such as 'ǎ', many dictionaries including this one use the Latin 'a' and place the tone at the end of the word. Here it is the third tone, so "hǎo" is written as "hao3". This makes it easier for those who only have US keyboards and no Unicode editor to still communicate in Pinyin.

This data format mismatch is not a big deal: just that somewhere along the line, between the original text dictionary and the display in the browser, a data massage has to be performed. Go templates allow this to be done by defining a custom template, so I chose that route. Alternatives could have been to do this as the dictionary is read in, or in the Javascript to display the final characters.

The code for the Pinyin formatter is given below. Please don't bother reading it unless you are *really* interested in knowing the rules for Pinyin formatting.

```
package pinyin

import (
    "io"
    "strings"
)

func PinyinFormatter(w io.Writer, format string, value ...interface{}) {
    line := value[0].(string)
    words := strings.Fields(line)
    for n, word := range words {
        // convert "u:" to "ü" if present
        uColon := strings.Index(word, "u:")
        if uColon != -1 {
            parts := strings.SplitN(word, "u:", 2)
            word = parts[0] + "ü" + parts[1]
        }
        println(word)
        // get last character, will be the tone if present
        chars := []rune(word)
        tone := chars[len(chars)-1]
        if tone == '5' {
            words[n] = string(chars[0 : len(chars)-1])
            println("lost accent on", words[n])
            continue
        }
        if tone < '1' || tone > '4' {
            continue
        }
        words[n] = addAccent(word, int(tone))
    }
    line = strings.Join(words, ` `)
    w.Write([]byte(line))
}

var (
    // maps 'a1' to '\u0101' etc
    aAccent = map[int]rune{
        '1': '\u0101',
        '2': '\u00e1',
        '3': '\u01ce', // '\u0103',
        '4': '\u00e0'}
    eAccent = map[int]rune{
        '1': '\u0113',
        '2': '\u00e9',
        '3': '\u011b', // '\u0115',
        '4': '\u00e8'}
    iAccent = map[int]rune{
        '1': '\u012b',
        '2': '\u00ed',
        '3': '\u01d0', // '\u012d',
        '4': '\u00ec'}
    oAccent = map[int]rune{
        '1': '\u014d',
        '2': '\u00f3',
        '3': '\u01d2', // '\u014f',
        '4': '\u00f2'}
    uAccent = map[int]rune{
        '1': '\u016b',
        '2': '\u00fa',
        '3': '\u01d4', // '\u016d',
        '4': '\u00f9'}
    üAccent = map[int]rune{
        '1': 'ǖ',
        '2': 'ǘ',
        '3': 'ǚ',
        '4': 'ǜ'}
)

func addAccent(word string, tone int) string {
    /*
     * Based on "Where do the tone marks go?"
     * at http://www.pinyin.info/rules/where.html
     */

    n := strings.Index(word, "a")
    if n != -1 {
        aAcc := aAccent[tone]
        // replace 'a' with its tone version
        word = word[0:n] + string(aAcc) + word[(n+1):len(word)-1]
    } else {
        n := strings.Index(word, "e")
        if n != -1 {
            eAcc := eAccent[tone]
            word = word[0:n] + string(eAcc) +
                word[(n+1):len(word)-1]
        } else {
            n = strings.Index(word, "ou")
            if n != -1 {
                oAcc := oAccent[tone]
                word = word[0:n] + string(oAcc) + "u" +
                    word[(n+2):len(word)-1]
            } else {
                chars := []rune(word)
                length := len(chars)
                // put tone onthe last vowel
            L:
                for n, _ := range chars {
                    m := length - n - 1
                    switch chars[m] {
                    case 'i':
                        chars[m] = iAccent[tone]
                        break L
                    case 'o':
                        chars[m] = oAccent[tone]
                        break L
                    case 'u':
                        chars[m] = uAccent[tone]
                        break L
                    case 'ü':
                        chars[m] = üAccent[tone]
                        break L
                    default:
                    }
                }
                word = string(chars[0 : len(chars)-1])
            }
        }
    }

    return word
}
```

How this is used is illustrated by the function `lookupWord`. This is called in response to an HTML Form request to find the English words in a dictionary.

```
func lookupWord(rw http.ResponseWriter, req *http.Request) {
        word := req.FormValue("word")
        words := d.LookupEnglish(word)

        pinyinMap := template.FormatterMap {"pinyin": pinyin.PinyinFormatter}
        t, err := template.ParseFile("html/DictionaryEntry.html", pinyinMap)
        if err != nil {
                http.Error(rw, err.String(), http.StatusInternalServerError)
                return
        }
        t.Execute(rw, words)
}
```

The HTML code is

```
<html>
  <body>
    <table border="1">
      <tr>
    <th>Word</th>
    <th>Traditional</th>
    <th>Simplified</th>
    <th>Pinyin</th>
    <th>English</th>
      </tr>
      {{with .Entries}}
      {{range .}}
      {.repeated section Entries}
      <tr>
    <td>{{.Word}}</td>
    <td>{{.Traditional}}</td>
    <td>{{.Simplified}}</td>
    <td>{{.Pinyin|pinyin}}</td>
    <td>
      <pre>
        {.repeated section Translations}
        {@|html}
        {.end}
      </pre>
    </td>
      </tr>
      {.end}
      {{end}}
      {{end}}
    </table>
  </body>
</html>
```

The Dictionary type

The text file containing the dictionary has lines of the form
traditional simplified [pinyin] /translation/translation/.../

For example,
好 好 [hao3] /good/well/proper/good to/easy to/very/so/(suffix indicating completion or readiness)/

We store each line as an `Entry` within the `Dictionary` package:

```
type Entry struct {
    Traditional string
    Simplified string
    Pinyin     string
    Translations []string
}
```

The dictionary itself is just an array of these entries:

```
type Dictionary struct {
    Entries []*Entry
}
```

Building the dictionary is easy enough. Just read each line and break the line into its various bits using simple string methods. Then add the line to the dictionary slice.

Looking up entries in this dictionary is straightforward: just search through until we find the appropriate key. There are about 100,000 entries in this dictionary: brute force by a linear search is fast enough. If it were necessary, faster storage and search mechanisms could easily be used.

The original dictionary grows by people on the Web adding in entries as they see fit. Consequently it isn't that well organised and contains repetitions and multiple entries. So looking up any word - either by Pinyin or by English - may return multiple matches. To cater for this, each lookup returns a "mini dictionary", just those lines in the full dictionary that match.

The Dictionary code is

```
package dictionary

import (
    "bufio"
    //"fmt"
    "os"
    "strings"
)

type Entry struct {
    Traditional  string
    Simplified   string
    Pinyin       string
    Translations []string
}

func (de Entry) String() string {
    str := de.Traditional + ` ` + de.Simplified + ` ` + de.Pinyin
    for _, t := range de.Translations {
        str = str + "\n    " + t
    }
    return str
}

type Dictionary struct {
    Entries []*Entry
}

func (d *Dictionary) String() string {
    str := ""
    for n := 0; n < len(d.Entries); n++ {
        de := d.Entries[n]
        str += de.String() + "\n"
    }
    return str
}

func (d *Dictionary) LookupPinyin(py string) *Dictionary {
    newD := new(Dictionary)
    v := make([]*Entry, 0, 100)
    for n := 0; n < len(d.Entries); n++ {
        de := d.Entries[n]
        if de.Pinyin == py {
            v = append(v, de)
        }
    }
    newD.Entries = v
    return newD
}

func (d *Dictionary) LookupEnglish(eng string) *Dictionary {
    newD := new(Dictionary)
    v := make([]*Entry, 0, 100)
    for n := 0; n < len(d.Entries); n++ {
        de := d.Entries[n]
        for _, e := range de.Translations {
            if e == eng {
                v = append(v, de)
            }
        }
    }
    newD.Entries = v
    return newD
}

func (d *Dictionary) LookupSimplified(simp string) *Dictionary {
    newD := new(Dictionary)
    v := make([]*Entry, 0, 100)

    for n := 0; n < len(d.Entries); n++ {
        de := d.Entries[n]
        if de.Simplified == simp {
            v = append(v, de)
        }
    }
    newD.Entries = v
    return newD
}

func (d *Dictionary) Load(path string) {

    f, err := os.Open(path)
    r := bufio.NewReader(f)
    if err != nil {
        println(err.Error())
        os.Exit(1)
    }

    v := make([]*Entry, 0, 100000)
    numEntries := 0
    for {
        line, err := r.ReadString('\n')
        if err != nil {
            break
        }
        if line[0] == '#' {
            continue
        }
        // fmt.Println(line)
        trad, simp, pinyin, translations := parseDictEntry(line)

        de := Entry{
            Traditional:  trad,
            Simplified:   simp,
            Pinyin:       pinyin,
            Translations: translations}

        v = append(v, &de)
        numEntries++
    }
    // fmt.Printf("Num entries %d\n", numEntries)
    d.Entries = v
}

func parseDictEntry(line string) (string, string, string, []string) {
    // format is
    //    trad simp [pinyin] /trans/trans/.../
    tradEnd := strings.Index(line, " ")
    trad := line[0:tradEnd]
    line = strings.TrimSpace(line[tradEnd:])

    simpEnd := strings.Index(line, " ")
    simp := line[0:simpEnd]
    line = strings.TrimSpace(line[simpEnd:])

    pinyinEnd := strings.Index(line, "]")
    pinyin := line[1:pinyinEnd]
    line = strings.TrimSpace(line[pinyinEnd+1:])

    translations := strings.Split(line, "/")
    // includes empty at start and end, so
    translations = translations[1 : len(translations)-1]
    return trad, simp, pinyin, translations
}
```

Flash cards

Each individual flash card is of the type `Flashcard`

```
type FlashCard struct {
    Simplified string
    English    string
    Dictionary *dictionary.Dictionary
}
```

At present we only store the simplified character and the English translation for that character. We also have a `Dictionary` which will contain only one entry for the entry we will have chosen somewhere.

A set of flash cards is defined by the type

```
type FlashCards struct {
    Name      string
    CardOrder string
    ShowHalf  string
    Cards     []*FlashCard
}
```

where the `CardOrder` will be `"random"` or `"sequential"` and the `ShowHalf` will be `"RANDOM_HALF"` or `"ENGLISH_HALF"` or `"CHINESE_HALF"` to determine which half of a new card is shown first.

The code for flash cards has nothing novel in it. We get data from the client browser and use JSON to create an object from the form data, and store the set of flashcards as a JSON string.

The Complete Server

The complete server is

```
/* Server
 */

package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "regexp"
    "text/template"
)

import (
    "dictionary"
    "flashcards"
    "templatefuncs"
)

var d *dictionary.Dictionary

func main() {
    if len(os.Args) != 2 {
        fmt.Fprint(os.Stderr, "Usage: ", os.Args[0], ":port\n")
        os.Exit(1)
    }
    port := os.Args[1]

    // dictionaryPath := "/var/www/go/chinese/cedict_ts.u8"
    dictionaryPath := "cedict_ts.u8"
    d = new(dictionary.Dictionary)
    d.Load(dictionaryPath)
    fmt.Println("Loaded dict", len(d.Entries))

    http.HandleFunc("/", listFlashCards)
    //fileServer := http.FileServer("/var/www/go/chinese/jscript", "/jscript/")
    fileServer := http.StripPrefix("/jscript/", http.FileServer(http.Dir("jscript")))
    http.Handle("/jscript/", fileServer)
    // fileServer = http.FileServer("/var/www/go/chinese/html", "/html/")
    fileServer = http.StripPrefix("/html/", http.FileServer(http.Dir("html")))
    http.Handle("/html/", fileServer)

    http.HandleFunc("/wordlook", lookupWord)
    http.HandleFunc("/flashcards.html", listFlashCards)
    http.HandleFunc("/flashcardSets", manageFlashCards)
    http.HandleFunc("/searchWord", searchWord)
    http.HandleFunc("/addWord", addWord)
    http.HandleFunc("/newFlashCardSet", newFlashCardSet)
    // deliver requests to the handlers
    err := http.ListenAndServe(port, nil)
    checkError(err)
    // That's it!
}

func indexPage(rw http.ResponseWriter, req *http.Request) {
    index, _ := ioutil.ReadFile("html/index.html")
    rw.Write([]byte(index))
}

func lookupWord(rw http.ResponseWriter, req *http.Request) {
    word := req.FormValue("word")
    words := d.LookupEnglish(word)

    //t := template.New("PinyinTemplate")
    t := template.New("DictionaryEntry.html")
    t = t.Funcs(template.FuncMap{"pinyin": templatefuncs.PinyinFormatter})
    t, err := t.ParseFiles("html/DictionaryEntry.html")
    if err != nil {
        http.Error(rw, err.Error(), http.StatusInternalServerError)
        return
    }
    t.Execute(rw, words)
}

type DictPlus struct {
    *dictionary.Dictionary
    Word     string
    CardName string
}

func searchWord(rw http.ResponseWriter, req *http.Request) {
    word := req.FormValue("word")
    searchType := req.FormValue("searchtype")
    cardName := req.FormValue("cardname")

    var words *dictionary.Dictionary
    var dp []DictPlus
    if searchType == "english" {
        words = d.LookupEnglish(word)
        d1 := DictPlus{Dictionary: words, Word: word, CardName: cardName}
        dp = make([]DictPlus, 1)
        dp[0] = d1
    } else {
        words = d.LookupPinyin(word)
        numTrans := 0
        for _, entry := range words.Entries {
            numTrans += len(entry.Translations)
        }
        dp = make([]DictPlus, numTrans)
        idx := 0
        for _, entry := range words.Entries {
            for _, trans := range entry.Translations {
                dict := new(dictionary.Dictionary)
                dict.Entries = make([]*dictionary.Entry, 1)
                dict.Entries[0] = entry
                dp[idx] = DictPlus{
                    Dictionary: dict,
                    Word:       trans,
                    CardName:   cardName}
                idx++
            }
        }
    }

    //t := template.New("PinyinTemplate")
    t := template.New("ChooseDictionaryEntry.html")
    t = t.Funcs(template.FuncMap{"pinyin": templatefuncs.PinyinFormatter})
    t, err := t.ParseFiles("html/ChooseDictionaryEntry.html")
    if err != nil {
        fmt.Println(err.Error())
        http.Error(rw, err.Error(), http.StatusInternalServerError)
        return
    }
    t.Execute(rw, dp)
}

func newFlashCardSet(rw http.ResponseWriter, req *http.Request) {
    defer http.Redirect(rw, req, "http:/flashcards.html", 200)

    newSet := req.FormValue("NewFlashcard")
    fmt.Println("New cards", newSet)
    // check against nasties:
    b, err := regexp.Match("[/$~]", []byte(newSet))
    if err != nil {
        return
    }
    if b {
        fmt.Println("No good string")
        return
    }

    flashcards.NewFlashCardSet(newSet)
    return
}

func addWord(rw http.ResponseWriter, req *http.Request) {
    url := req.URL
    fmt.Println("url", url.String())
    fmt.Println("query", url.RawQuery)

    word := req.FormValue("word")
    cardName := req.FormValue("cardname")
    simplified := req.FormValue("simplified")
    pinyin := req.FormValue("pinyin")
    traditional := req.FormValue("traditional")
    translations := req.FormValue("translations")

    fmt.Println("word is ", word, " card is ", cardName,
        " simplified is ", simplified, " pinyin is ", pinyin,
        " trad is ", traditional, " trans is ", translations)
    flashcards.AddFlashEntry(cardName, word, pinyin, simplified,
        traditional, translations)
    // add another card?
    addFlashCards(rw, cardName)
}

func listFlashCards(rw http.ResponseWriter, req *http.Request) {

    flashCardsNames := flashcards.ListFlashCardsNames()
    t, err := template.ParseFiles("html/ListFlashcards.html")
    if err != nil {
        http.Error(rw, err.Error(), http.StatusInternalServerError)
        return
    }
    t.Execute(rw, flashCardsNames)
}

/*
 * Called from ListFlashcards.html on form submission
 */
func manageFlashCards(rw http.ResponseWriter, req *http.Request) {

    set := req.FormValue("flashcardSets")
    order := req.FormValue("order")
    action := req.FormValue("submit")
    half := req.FormValue("half")
    fmt.Println("set chosen is", set)
    fmt.Println("order is", order)
    fmt.Println("action is", action)

    cardname := "flashcardSets/" + set

    //components := strings.Split(req.URL.Path[1:], "/", -1)
    //cardname := components[1]
    //action := components[2]
    fmt.Println("cardname", cardname, "action", action)
    if action == "Show cards in set" {
        showFlashCards(rw, cardname, order, half)
    } else if action == "List words in set" {
        listWords(rw, cardname)
    } else if action == "Add cards to set" {
        addFlashCards(rw, set)
    }
}

func showFlashCards(rw http.ResponseWriter, cardname, order, half string) {
    fmt.Println("Loading card name", cardname)
    cards := new(flashcards.FlashCards)
    //cards.Load(cardname, d)
    //flashcards.SaveJSON(cardname + ".json", cards)
    flashcards.LoadJSON(cardname, &cards)
    if order == "Sequential" {
        cards.CardOrder = "SEQUENTIAL"
    } else {
        cards.CardOrder = "RANDOM"
    }
    fmt.Println("half is", half)
    if half == "Random" {
        cards.ShowHalf = "RANDOM_HALF"
    } else if half == "English" {
        cards.ShowHalf = "ENGLISH_HALF"
    } else {
        cards.ShowHalf = "CHINESE_HALF"
    }
    fmt.Println("loaded cards", len(cards.Cards))
    fmt.Println("Card name", cards.Name)

    //t := template.New("PinyinTemplate")
    t := template.New("ShowFlashcards.html")
    t = t.Funcs(template.FuncMap{"pinyin": templatefuncs.PinyinFormatter})
    t, err := t.ParseFiles("html/ShowFlashcards.html")
    if err != nil {
        fmt.Println(err.Error())
        http.Error(rw, err.Error(), http.StatusInternalServerError)
        return
    }
    err = t.Execute(rw, cards)
    if err != nil {
        fmt.Println("Execute error " + err.Error())
        http.Error(rw, err.Error(), http.StatusInternalServerError)
        return
    }
}

func listWords(rw http.ResponseWriter, cardname string) {
    fmt.Println("Loading card name", cardname)
    cards := new(flashcards.FlashCards)
    //cards.Load(cardname, d)
    flashcards.LoadJSON(cardname, cards)
    fmt.Println("loaded cards", len(cards.Cards))
    fmt.Println("Card name", cards.Name)

    //t := template.New("PinyinTemplate")
    t := template.New("ListWords.html")
    if t.Tree == nil || t.Root == nil {
        fmt.Println("New t is an incomplete or empty template")
    }
    t = t.Funcs(template.FuncMap{"pinyin": templatefuncs.PinyinFormatter})
    t, err := t.ParseFiles("html/ListWords.html")
    if t.Tree == nil || t.Root == nil {
        fmt.Println("Parsed t is an incomplete or empty template")
    }

    if err != nil {
        fmt.Println("Parse error " + err.Error())
        http.Error(rw, err.Error(), http.StatusInternalServerError)
        return
    }
    err = t.Execute(rw, cards)
    if err != nil {
        fmt.Println("Execute error " + err.Error())
        http.Error(rw, err.Error(), http.StatusInternalServerError)
        return
    }
    fmt.Println("No error ")
```

```
}

func addFlashCards(rw http.ResponseWriter, cardname string) {
    t, err := template.ParseFiles("html/AddWordToSet.html")
    if err != nil {
        fmt.Println("Parse error " + err.Error())
        http.Error(rw, err.Error(), http.StatusInternalServerError)
        return
    }
    cards := flashcards.GetFlashCardsByName(cardname, d)
    t.Execute(rw, cards)
    if err != nil {
        fmt.Println("Execute error " + err.Error())
        http.Error(rw, err.Error(), http.StatusInternalServerError)
        return
    }

}

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

Other Bits: JavaScript and CSS

On request, a set of flashcards will be loaded into the browser. A much abbreviated set is shown below. The display of these cards is controlled by JavaScript and CSS files. These aren't relevant to the Go server so are omitted. Those interested can download the code.

```
<html>
  <head>
    <title>
      Flashcards for Common Words
    </title>

    <link type="text/css" rel="stylesheet"
          href="/html/CardStylesheet.css">
    </link>

    <script type="text/javascript"
            language="JavaScript1.2" src="/jscript/jquery.js">
      <!-- empty -->
    </script>

    <script type="text/javascript"
            language="JavaScript1.2" src="/jscript/slideviewer.js">
      <!-- empty -->
    </script>

    <script type="text/javascript"
            language="JavaScript1.2">
      cardOrder = RANDOM;
      showHalfCard = RANDOM_HALF;
    </script>
  </head>
  <body onload="showSlides();">
    <h1>
      Flashcards for Common Words
    </h1>
    <p>
        <div class="card">

      <div class="english">
        <div class="vcenter">
          hello
        </div>
      </div>

          <div class="pinyin">
        <div class="vcenter">
          nǐ hǎo
        </div>

          </div>

          <div class="traditional">
        <div class="vcenter">
          你好
        </div>
          </div>

          <div class="simplified">
        <div class="vcenter">
          你好
        </div>

          </div>

          <div class ="translations">
        <div class="vcenter">
          hello <br />
          hi <br />
          how are you? <br />
        </div>

              </div>
        </div>
        <div class="card">
      <div class="english">
        <div class="vcenter">
          hello (interj., esp. on telephone)
        </div>
      </div>

          <div class="pinyin">

        <div class="vcenter">
          wèi
        </div>
          </div>

          <div class="traditional">
        <div class="vcenter">
          喂
        </div>
          </div>

          <div class="simplified">

        <div class="vcenter">
          喂
        </div>
          </div>

          <div class ="translations">
        <div class="vcenter">
          hello (interj., esp. on telephone) <br />
          hey <br />

          to feed (sb or some animal) <br />
        </div>
              </div>
        </div>
```

```
      </p>

      <p class ="return">
        Press &lt;Space&gt; to continue
      <br/>
        <a href="http:/flashcards.html"> Return to Flash Cards list</a>
      </p>
    </body>
  </html>
```

Chapter 11 HTML

The Web was originally created to serve HTML documents. Now it is used to serve all sorts of documents as well as data of different kinds. Nevertheless, HTML is still the main document type delivered over the Web. Go has basic mechanisms for parsing HTML documents, which are covered in this chapter.

... in progress

Chapter 12 XML

XML is a significant markup language mainly intended as a means of serialising data structures as a text document. Go has basic support for XML document processing.

Introduction

XML is now a widespread way of representing complex data structures serialised into text format. It is used to describe documents such as DocBook and XHTML. It is used in specialised markup languages such as MathML and CML (Chemistry Markup Language). It is used to encode data as SOAP messages for Web Services, and the Web Service can be specified using WSDL (Web Services Description Language).

At the simplest level, XML allows you to define your own tags for use in text documents. Tags can be nested and can be interspersed with text. Each tag can also contain attributes with values. For example,

```
<person>
  <name>
    <family> Newmarch </family>
    <personal> Jan </personal>
  </name>
  <email type="personal">
    jan@newmarch.name
  </email>
  <email type="work">
    j.newmarch@boxhill.edu.au
  </email>
</person>
```

The structure of any XML document can be described in a number of ways:

- A document type definition DTD is good for describing structure
- XML schema are good for describing the data types used by an XML document
- RELAX NG is proposed as an alternative to both

There is argument over the relative value of each way of defining the structure of an XML document. We won't buy into that, as Go does not suport any of them. Go cannot check for validity of any document against a schema, but only for well-formedness.

Four topics are discussed in this chapter: parsing an XML stream, marshalling and unmarshalling Go data into XML, and XHTML.

Parsing XML

Go has an XML parser which is created using `NewParser`. This takes an `io.Reader` as parameter and returns a pointer to `Parser`. The main method of this type is `Token` which returns the next token in the input stream. The token is one of the types `StartElement`, `EndElement`, `CharData`, `Comment`, `ProcInst` or `Directive`.

The types are

`StartElement`

The type `StartElement` is a structure with two field types:

```
type StartElement struct {
    Name Name
    Attr []Attr
}

type Name struct {
    Space, Local string
}

type Attr struct {
    Name  Name
    Value string
}
```

`EndElement`

This is also a structure

```
type EndElement struct {
    Name Name
}
```

`CharData`

This type represents the text content enclosed by a tag and is a simple type

```
type CharData []byte
```

`Comment`

Similarly for this type

`type Comment []byte`

`ProcInst`

A ProcInst represents an XML processing instruction of the form `<?target inst?>`

```
type ProcInst struct {
    Target string
    Inst   []byte
}
```

`Directive`

A Directive represents an XML directive of the form <!text>. The bytes do not include the <! and > markers.

```
type Directive []byte
```

A program to print out the tree structure of an XML document is

```
/* Parse XML
 */

package main

import (
    "encoding/xml"
    "fmt"
    "io/ioutil"
    "os"
    "strings"
)

func main() {
    if len(os.Args) != 2 {
        fmt.Println("Usage: ", os.Args[0], "file")
        os.Exit(1)
    }
    file := os.Args[1]
    bytes, err := ioutil.ReadFile(file)
    checkError(err)
    r := strings.NewReader(string(bytes))

    parser := xml.NewDecoder(r)
    depth := 0
    for {
        token, err := parser.Token()
        if err != nil {
            break
        }
        switch t := token.(type) {
        case xml.StartElement:
            elmt := xml.StartElement(t)
            name := elmt.Name.Local
            printElmt(name, depth)
            depth++
        case xml.EndElement:
            depth--
            elmt := xml.EndElement(t)
            name := elmt.Name.Local
            printElmt(name, depth)
        case xml.CharData:
            bytes := xml.CharData(t)
            printElmt("\""+string([]byte(bytes))+"\"", depth)
        case xml.Comment:
            printElmt("Comment", depth)
        case xml.ProcInst:
            printElmt("ProcInst", depth)
        case xml.Directive:
            printElmt("Directive", depth)
        default:
            fmt.Println("Unknown")
        }
    }
}

func printElmt(s string, depth int) {
    for n := 0; n < depth; n++ {
        fmt.Print("  ")
    }
    fmt.Println(s)
}

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

Note that the parser includes all CharData, including the whitespace between tags.

If we run this program against the `person` data structure given earlier, it produces

```
person
  "
  "
  name
    "
    "
    family
      " Newmarch "
    family
    "
    "
    personal
      " Jan "
    personal
    "
  "
  name
  "
  "
  email
    "
    jan@newmarch.name
  "
  email
  "
  "
  email
    "
    j.newmarch@boxhill.edu.au
  "
  email
  "
"
person
"
"
```

Note that as no DTD or other XML specification has been used, the tokenizer correctly prints out all the white space (a DTD may specify that the whitespace can be ignored, but without it that assumption cannot be made.)

There is a potential trap in using this parser. It re-uses space for strings, so that once you see a token you need to copy its value if you want to refer to it later. Go has methods such as `func (c CharData) Copy() CharData` to make a copy of data.

Unmarshalling XML

Go provides a function `Unmarshal` and a method `func (*Parser) Unmarshal` to unmarshal XML into Go data structures. The unmarshalling is not perfect: Go and XML are different languages.

We consider a simple example before looking at the details. We take the XML document given earlier of

```
<person>
  <name>
    <family> Newmarch </family>
    <personal> Jan </personal>
  </name>
  <email type="personal">
    jan@newmarch.name
  </email>
  <email type="work">
    j.newmarch@boxhill.edu.au
  </email>
</person>
```

We would like to map this onto the Go structures

```
type Person struct {
    Name Name
    Email []Email
}

type Name struct {
    Family string
    Personal string
}

type Email struct {
    Type string
    Address string
}
```

This requires several comments:

1. Unmarshalling uses the Go reflection package. This requires that all fields by public i.e. start with a capital letter. Earlier versions of Go used case-insensitive matching to match fields such as the XML string "name" to the field `Name`. Now, though, *case-sensitive* matching is used. To perform a match, the structure fields must be tagged to show the XML string that will be matched against. This changes `Person` to

```
type Person struct {
    Name Name `xml:"name"`
    Email []Email `xml:"email"`
}
```

1. While tagging of fields can attach XML strings to fields, it can't do so with the names of the structures. An additional field is required, with field name "XMLName". This only affects the top-level struct, `Person`

```
type Person struct {
    XMLName Name `xml:"person"`
    Name Name `xml:"name"`
    Email []Email `xml:"email"`
}
```

1. Repeated tags in the map to a slice in Go
2. Attributes within tags will match to fields in a structure only if the Go field has the tag ",attr". This occurs with the field `Type` of `Email`, where matching the attribute "type" of the "email" tag requires `` `xml:"type,attr"` ``
3. If an XML tag has no attributes and only has character data, then it matches a `string` field by the same name (case-sensitive, though). So the tag `` `xml:"family"` `` with character data "Newmarch" maps to the string field `Family`
4. But if the tag has attributes, then it must map to a structure. Go assigns the character data to the field with tag `,chardata`. This occurs with the "email" data and the field `Address` with tag `,chardata`

A program to unmarshal the document above is

```
/* Unmarshal
 */

package main

import (
    "encoding/xml"
    "fmt"
    "os"
    //"strings"
)

type Person struct {
    XMLName Name    `xml:"person"`
    Name    Name    `xml:"name"`
    Email   []Email `xml:"email"`
}

type Name struct {
    Family   string `xml:"family"`
    Personal string `xml:"personal"`
}

type Email struct {
    Type    string `xml:"type,attr"`
    Address string `xml:",chardata"`
}
```

```
func main() {
    str := `<?xml version="1.0" encoding="utf-8"?>
<person>
  <name>
    <family> Newmarch </family>
    <personal> Jan </personal>
  </name>
  <email type="personal">
    jan@newmarch.name
  </email>
  <email type="work">
    j.newmarch@boxhill.edu.au
  </email>
</person>`

    var person Person

    err := xml.Unmarshal([]byte(str), &person)
    checkError(err)

    // now use the person structure e.g.
    fmt.Println("Family name: \"" + person.Name.Family + "\"")
    fmt.Println("Second email address: \"" + person.Email[1].Address + "\"")
}

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

(Note the spaces are correct.). The strict rules are given in the package specification.

Marshalling XML

Go 1 also has support for marshalling data structures into an XML document. The function is

```
func Marshal(v interface}{) ([]byte, error)
```

This was used as a check in the last two lines of the previous program.

XHTML and HTML

XHTML

HTML does not conform to XML syntax. It has unterminated tags such as `<br>`. XHTML is a cleanup of HTML to make it compliant to XML. Documents in XHTML can be managed using the techniques above for XML.

HTML

There is some support in the XML package to handle HTML documents even though they are not XML-compliant. The XML parser discussed earlier can handle many HTML documents if it is modified by

```
parser := xml.NewDecoder(r)
parser.Strict = false
parser.AutoClose = xml.HTMLAutoClose
parser.Entity = xml.HTMLEntity
```

Conclusion

Go has basic support for dealing with XML strings. It does not as yet have mechanisms for dealing with XML specification languages such as XML Schema or Relax NG.

Chapter 13 Remote Procedure Call

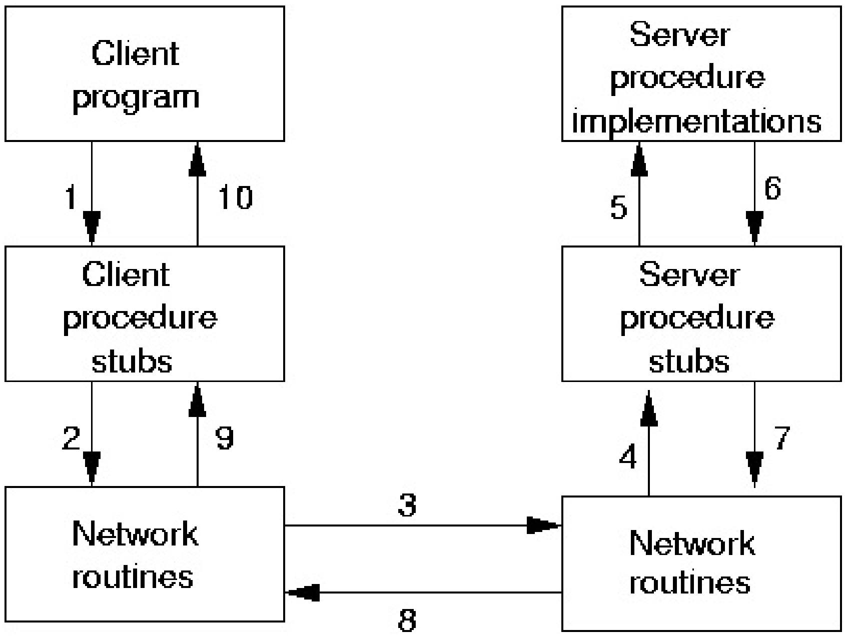
Introduction

Socket and HTTP programming use a message-passing paradigm. A client sends a message to a server which usually sends a message back. Both sides ae responsible for creating messages in a format understood by both sides, and in reading the data out of those messages.

However, most standalone applications do not make so much use of message passing techniques. Generally the preferred mechanism is that of the function (or method or procedure) call. In this style, a program will call a function with a list of parameters, and on completion of the function call will have a set of return values. These values may be the function value, or if addresses have been passed as parameters then the contents of those addresses might have been changed.

The remote procedure call is an attempt to bring this style of programming into the network world. Thus a client will make what looks to it like a normal procedure call. The client-side will package this into a network message and transfer it to the server. The server will unpack this and turn it back into a procedure call on the server side. The results of this call will be packaged up for return to the client.

Diagrammatically it looks like



where the steps are

1. The client calls the local stub procedure. The stub packages up the parameters into a network message. This is called marshalling.
2. Networking functions in the O/S kernel are called by the stub to send the message.
3. The kernel sends the message(s) to the remote system. This may be connection-oriented or connectionless.
4. A server stub unmarshalls the arguments from the network message.

5. The server stub executes a local procedure call.
6. The procedure completes, returning execution to the server stub.
7. The server stub marshals the return values into a network message.
8. The return messages are sent back.
9. The client stub reads the messages using the network functions.
10. The message is unmarshalled. and the return values are set on the stack for the local process.

There are two common styles for implementing RPC. The first is typified by Sun's RPC/ONC and by CORBA. In this, a specification of the service is given in some abstract language such as CORBA IDL (interface definition language). This is then compiled into code for the client and for the server. The client then writes a normal program containing calls to a procedure/function/method which is linked to the generated client-side code. The server-side code is actually a server itself, which is linked to the procedure implementation that you write.

In this way, the client-side code is almost identical in appearance to a normal procedure call. Generally there is a little extra code to locate the server. In Sun's ONC, the address of the server must be known; in CORBA a naming service is called to find the address of the server; In Java RMI, the IDL is Java itself and a naming service is used to find the address of the service.

In the second style, you have to make use of a special client API. You hand the function name and its parameters to this library on the client side. On the server side, you have to explicitly write the server yourself, as well as the remote procedure implementation.

This approach is used by many RPC systems, such as Web Services. It is also the approach used by Go's RPC.

Go RPC

Go's RPC is so far unique to Go. It is different to the other RPC systems, so a Go client will only talk to a Go server. It uses the Gob serialisation system discussed in "Chapter 4 Data serialisation - The gob package", which defines the data types which can be used.

RPC systems generally make some restrictions on the functions that can be called across the network. This is so that the RPC system can properly determine what are value arguments to be sent, what are reference arguments to receive answers, and how to signal errors.

In Go, the restriction is that

- the function must be public (begin with a capital letter);
- have exactly two arguments, the first is a pointer to value data to be received by the function from the client, and the second is a pointer to hold the answers to be returned to the client; and
- have a return value of type `error`

For example, a valid function is

```
F(&T1, &T2) error
```

The restriction on arguments means that you typically have to define a structure type. Go's RPC uses the `gob` package for marshalling and unmarshalling data, so the argument types have to follow the rules of `gob` as discussed in an earlier chapter.

We shall follow the example given in the Go documentation, as this illustrates the important points. The server performs two operations which are trivial - they do not require the "grunt" of RPC, but are simple to understand. The two operations are to multiply two integers, and the second is to find the quotient and remainder after dividing the first by the second.

The two values to be manipulated are given in a structure:

```
type Args struct {
    A, B int
}
```

The sum is just an `int`, while the quotient/remainder is another structure

```
type Quotient struct {
    Quo, Rem int
}
```

We will have two functions, multiply and divide to be callable on the RPC server. These functions will need to be registered with the RPC system. The function `Register` takes a single parameter, which is an interface. So we need a type with these two functions:

```
type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return error.String("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}
```

The underlying type of `Arith` is given as `int`. That doesn't matter - any type could have done.

An object of this type can now be registered using `Register`, and then its methods can be called by the RPC system.

HTTP RPC Server

Any RPC needs a transport mechanism to get messages across the network. Go can use HTTP or TCP. The advantage of the HTTP mechanism is that it can leverage off the HTTP suport library. You need to add an RPC handler to the HTTP layer which is done using `HandleHTTP` and then start an HTTP server. The complete code is

```
/**
* ArithServer
 */

package main

import (
    "fmt"
    "net/rpc"
    "errors"
    "net/http"
)

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}

func main() {

    arith := new(Arith)
    rpc.Register(arith)
    rpc.HandleHTTP()

    err := http.ListenAndServe(":1234", nil)
    if err != nil {
        fmt.Println(err.Error())
    }
}
```

HTTP RPC client

The client needs to set up an HTTP connection to the RPC server. It needs to prepare a structure with the values to be sent, and the address of a variable to store the results in. Then it can make a `Call` with arguments:

- The name of the remote function to execute
- The values to be sent
- The address of a variable to store the result in

A client that calls both functions of the arithmetic server is

```
/**
* ArithClient
 */

package main

import (
    "net/rpc"
    "fmt"
    "log"
    "os"
)

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

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

    client, err := rpc.DialHTTP("tcp", serverAddress+":1234")
    if err != nil {
        log.Fatal("dialing:", err)
    }
    // Synchronous call
    args := Args{17, 8}
    var reply int
    err = client.Call("Arith.Multiply", args, &reply)
    if err != nil {
        log.Fatal("arith error:", err)
    }
    fmt.Printf("Arith: %d*%d=%d\n", args.A, args.B, reply)

    var quot Quotient
    err = client.Call("Arith.Divide", args, &quot)
    if err != nil {
        log.Fatal("arith error:", err)
    }
    fmt.Printf("Arith: %d/%d=%d remainder %d\n", args.A, args.B, quot.Quo, quot.Rem)

}
```

TCP RPC server

A version of the server that uses TCP sockets is

```
/**
* TCPArithServer
 */

package main

import (
    "fmt"
    "net/rpc"
    "errors"
    "net"
    "os"
)

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}

func main() {

    arith := new(Arith)
    rpc.Register(arith)

    tcpAddr, err := net.ResolveTCPAddr("tcp", ":1234")
    checkError(err)

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

    /* This works:
    rpc.Accept(listener)
    */
    /* and so does this:
     */
    for {
        conn, err := listener.Accept()
        if err != nil {
            continue
        }
        rpc.ServeConn(conn)
    }

}

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

Note that the call to `Accept` is blocking, and just handles client connections. If the server wishes to do other work as well, it should call this in a goroutine.

TCP RPC client

A client that uses the TCP server and calls both functions of the arithmetic server is

```
/**
* TCPArithClient
 */

package main

import (
    "net/rpc"
    "fmt"
    "log"
    "os"
)

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

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

    client, err := rpc.Dial("tcp", service)
    if err != nil {
        log.Fatal("dialing:", err)
    }
    // Synchronous call
    args := Args{17, 8}
    var reply int
    err = client.Call("Arith.Multiply", args, &reply)
    if err != nil {
        log.Fatal("arith error:", err)
    }
    fmt.Printf("Arith: %d*%d=%d\n", args.A, args.B, reply)

    var quot Quotient
    err = client.Call("Arith.Divide", args, &quot)
    if err != nil {
        log.Fatal("arith error:", err)
    }
    fmt.Printf("Arith: %d/%d=%d remainder %d\n", args.A, args.B, quot.Quo, quot.Rem)

}
```

Matching values

We note that the types of the value arguments are not the same on the client and server. In the server, we have used `Values` while in the client we used `Args`. That doesn't matter, as we are following the rules of `gob` serialisation, and the names an types of the two structures' fields match. Better programming practise would say that the names should be the same!

However, this does point out a possible trap in using Go RPC. If we change the structure in the client to be, say,

```
type Values struct {
    C, B int
}
```

then `gob` has no problems: on the server-side the unmarshalling will ignore the value of C given by the client, and use the default zero value for A.

Using Go RPC will require a rigid enforcement of the stability of field names and types by the programmer. We note that there is no version control mechanism to do this, and no mechanism in `gob` to signal any possible mismatches.

JSON

This section adds nothing new to the earlier concepts. It just uses a different "wire" format for the data, JSON instead of `gob`. As such, clients or servers could be written in other languages that understand sockets and JSON.

JSON RPC client

A client that calls both functions of the arithmetic server is

```
/* JSONArithCLient
 */

package main

import (
    "net/rpc/jsonrpc"
    "fmt"
    "log"
    "os"
)

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

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

    client, err := jsonrpc.Dial("tcp", service)
    if err != nil {
        log.Fatal("dialing:", err)
    }
    // Synchronous call
    args := Args{17, 8}
    var reply int
    err = client.Call("Arith.Multiply", args, &reply)
    if err != nil {
        log.Fatal("arith error:", err)
    }
    fmt.Printf("Arith: %d*%d=%d\n", args.A, args.B, reply)

    var quot Quotient
    err = client.Call("Arith.Divide", args, &quot)
    if err != nil {
        log.Fatal("arith error:", err)
    }
    fmt.Printf("Arith: %d/%d=%d remainder %d\n", args.A, args.B, quot.Quo, quot.Rem)

}
```

JSON RPC server

A version of the server that uses JSON encoding is

```
/* JSONArithServer
 */

package main

import (
    "fmt"
    "net/rpc"
    "net/rpc/jsonrpc"
    "os"
    "net"
    "errors"
)
//import ("fmt"; "rpc"; "os"; "net"; "log"; "http")

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}

func main() {

    arith := new(Arith)
    rpc.Register(arith)
    tcpAddr, err := net.ResolveTCPAddr("tcp", ":1234")
    checkError(err)

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

    /* This works:
    rpc.Accept(listener)
    */
    /* and so does this:
     */
    for {
        conn, err := listener.Accept()
        if err != nil {
            continue
        }
        jsonrpc.ServeConn(conn)
    }

}

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

Conclusion

RPC is a popular means of distributing applications. Several ways of doing it have been presented here. What is missing from Go is support for the currently fashionable (but extremely badly enginereed) SOAP RPC mechanism.

Chapter 14 Network channels

Warning

The `netchan` package is being reworked. While it was in earlier versions of Go, it is not in Go 1. It is available in the `old/netchan` package if you still need it. This chapter describes this old version. Do not use it for new code.

Introduction

There are many models for sharing information between communicating processes. One of the more elegant is Here'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 listen 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

```go
/* 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."

```go
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 golang.org has an example of multiplexing, where channels of channels are used. The idea is that instead of sharing one channel, a new communicator is given their own channel to have a private 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 communication.

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.

Chapter 15 Web sockets

Web sockets are designed to answer a common problem with web systems: the server is unable to initiate or push content to a user agent such as a browser. Web sockets allow a full duplex connection to be established to allow this. Go has nearly complete support for them.

Warning

The Web Sockets package is not currently in the main Go 1.x tree and is not included in the current distributions. To use it, you need to install it by

```
go get golang.org/x/net/websocket
```

Introduction

(Note: The websockets model changed in release r61 of Go 1.0. This describes the current package.)

The standard model of interaction between a web user agent such as a browser and a web server such as Apache is that the user agent makes HTTP requests and the server makes a single reply to each one. In the case of a browser, the request is made by clicking on a link, entering a URL into the address bar, clicking on the forward or back buttons, etc. The response is treated as a new page and is loaded into a browser window.

This traditional model has many drawbacks. The first is that each request opens and closes a new TCP connection. HTTP 1.1 solved this by allowing persistent connections, so that a connection could be held open for a short period to allow for multiple requests (e.g. for images) to be made on the same server.

While HTTP 1.1 persistent connections alleviate the problem of slow loading of a page with many graphics, it does not improve the interaction model. Even with forms, the model is still that of submitting the form and displaying the response as a new page. JavaScript helps in allowing error checking to be performed on form data before submission, but does not change the model.

AJAX (Asynchronous JavaScript and XML) made a significant advance to the user interaction model. This allows a browser to make a request and just use the response to update the display in place using the HTML Document Object Model (DOM). But again the interaction model is the same. AJAX just affects how the browser manages the returned pages. There is no explicit extra support in Go for AJAX, as none is needed: the HTTP server just sees an ordinary HTTP POST request with possibly some XML or JSON data, and this can be dealt with using techniques already discussed.

All of these are still browser to server communication. What is missing is server initiated communications to the browser. This can be filled by Web sockets: the browser (or any user agent) keeps open a long-lived TCP connection to a Web sockets server. The TCP connection allows either side to send arbitrary packets, so any application protocol can be used on a web socket.

How a websocket is started is by the user agent sending a special HTTP request that says "switch to web sockets". The TCP connection underlying the HTTP request is kept open, but both user agent and server switch to using the web sockets protocol instead of getting an HTTP response and closing the socket.

Note that it is still the browser or user agent that initiates the Web socket connection. The browser does not run a TCP server of its own. While the specification is complex, the protocol is designed to be fairly easy to use. The client opens an HTTP connection and then replaces the HTTP protocol with its own WS protocol, re-using the same TCP connection.

Web socket server

A web socket server starts off by being an HTTP server, accepting TCP conections and handling the HTTP requests on the TCP connection. When a request comes in that switches that connection to a being a web socket connection, the protocol handler is changed from an HTTP handler to a WebSocket handler. So it is only that TCP connection that gets its role changed: the server continues to be an HTTP server for other requests, while the TCP socket underlying that one connection is used as a web socket.

One of the simple HTTP servers we discussed in Chapter 8: HTTP, registered varous handlers such as a file handler or a function handler. To handle web socket requests we simply register a different type of handler - a web socket handler. Which handler the server uses is based on the URL pattern. For example, a file handler might be registered for "/", a function handler for "/cgi-bin/..." and a web sockets handler for "/ws".

An HTTP server that is only expecting to be used for web sockets might run by

```
func main() {
    http.Handle("/", websocket.Handler(WSHandler))
    err := http.ListenAndServe(":12345", nil)
    checkError(err)
}
```

A more complex server might handle both HTTP and web socket requests simply by adding in more handlers.

The Message object

HTTP is a stream protocol. Web sockets are frame-based. You prepare a block of data (of any size) and send it as a set of frames. Frames can contain either strings in UTF-8 encoding or a sequence of bytes.

The simplest way of using web sockets is just to prepare a block of data and ask the Go websocket library to package it as a set of frame data, send them across the wire and receive it as the same block. The `websocket` package contains a convenience object `Message` to do just that. The `Message` object has two methods, `Send` and `Receive` which take a websocket as first parameter. The second parameter is either the address of a variable to store data in, or the data to be sent. Code to send string data would look like

```
msgToSend := "Hello"
err := websocket.Message.Send(ws, msgToSend)

var msgToReceive string
err := websocket.Message.Receive(conn, &msgToReceive)
```

Code to send byte data would look like

```
dataToSend := []byte{0, 1, 2}
err := websocket.Message.Send(ws, dataToSend)

var dataToReceive []byte
err := websocket.Message.Receive(conn, &dataToReceive)
```

An echo server to send and receive string data is given below. Note that in web sockets either side can initiate sending of messages, and in this server we send messages from the server to a client when it connects (send/receive) instead of the more normal receive/send server. The server is

```
/* EchoServer
 */
package main

import (
    "fmt"
    "net/http"
    "os"
    // "io"
    "golang.org/x/net/websocket"
)

func Echo(ws *websocket.Conn) {
    fmt.Println("Echoing")

    for n := 0; n < 10; n++ {
        msg := fmt.Sprintf("Hello %d", n)
        fmt.Println("Sending to client: " + msg)
        err := websocket.Message.Send(ws, msg)
        if err != nil {
            fmt.Println("Can't send")
            break
        }

        var reply string
        err = websocket.Message.Receive(ws, &reply)
        if err != nil {
            fmt.Println("Can't receive")
            break
        }
        fmt.Println("Received back from client: " + reply)
    }
}

func main() {

    http.Handle("/", websocket.Handler(Echo))
    err := http.ListenAndServe(":12345", nil)
    checkError(err)
}

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

A client that talks to this server is

```
/* EchoClient
 */
package main

import (
    "golang.org/x/net/websocket"
    "fmt"
    "io"
    "os"
)

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

    conn, err := websocket.Dial(service, "", "http://localhost")
    checkError(err)
    var msg string
    for {
        err := websocket.Message.Receive(conn, &msg)
        if err != nil {
            if err == io.EOF {
                // graceful shutdown by server
                break
            }
            fmt.Println("Couldn't receive msg " + err.Error())
            break
        }
        fmt.Println("Received from server: " + msg)
        // return the msg
        err = websocket.Message.Send(conn, msg)
        if err != nil {
            fmt.Println("Couldn't return msg")
            break
        }
    }
    os.Exit(0)
}

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

The url for the client running on the same machine as the server should be `ws://localhost:12345/`

The JSON object

It is expected that many websocket clients and servers will exchange data in JSON format. For Go programs this means that a Go object will be marshalled into JSON format as described in Chapter 4: Serialisation and then sent as a UTF-8 string, while the receiver will read this string and unmarshal it back into a Go object.

The `websocket` convenience object `JSON` will do this for you. It has methods `Send` and `Receive` for sending and receiving data, just like the `Message` object.

A client that sends a `Person` object in JSON format is

```go
/* PersonClientJSON
 */
package main

import (
    "golang.org/x/net/websocket"
    "fmt"
    "os"
)

type Person struct {
    Name   string
    Emails []string
}

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

    conn, err := websocket.Dial(service, "",
        "http://localhost")
    checkError(err)

    person := Person{Name: "Jan",
        Emails: []string{"ja@newmarch.name", "jan.newmarch@gmail.com"},
    }

    err = websocket.JSON.Send(conn, person)
    if err != nil {
        fmt.Println("Couldn't send msg " + err.Error())
    }
    os.Exit(0)
}

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

and a server that reads it is

```
/* PersonServerJSON
 */
package main

import (
    "golang.org/x/net/websocket"
    "fmt"
    "net/http"
    "os"
)

type Person struct {
    Name   string
    Emails []string
}

func ReceivePerson(ws *websocket.Conn) {
    var person Person
    err := websocket.JSON.Receive(ws, &person)
    if err != nil {
        fmt.Println("Can't receive")
    } else {

        fmt.Println("Name: " + person.Name)
        for _, e := range person.Emails {
            fmt.Println("An email: " + e)
        }
    }
}

func main() {

    http.Handle("/", websocket.Handler(ReceivePerson))
    err := http.ListenAndServe(":12345", nil)
    checkError(err)
}

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

The Codec type

The `Message` and `JSON` objects are both instances of the type `Codec`. This type is defined by

```
type Codec struct {
    Marshal   func(v interface{}) (data []byte, payloadType byte, err error)
    Unmarshal func(data []byte, payloadType byte, v interface{}) (err error)
}
```

The type `Codec` implements the `Send` and `Receive` methods used earlier.

It is likely that websockets will also be used to exchange XML data. We can build an XML `Codec` object by wrapping the XML marshal and unmarshal methods discussed in Chapter 12: XML to give a suitable `Codec` object.

We can create a `XMLCodec` package in this way:

```
package xmlcodec

import (
    "encoding/xml"
    "golang.org/x/net/websocket"
)

func xmlMarshal(v interface{}) (msg []byte, payloadType byte, err error) {
    //buff := &bytes.Buffer{}
    msg, err = xml.Marshal(v)
    //msgRet := buff.Bytes()
    return msg, websocket.TextFrame, nil
}

func xmlUnmarshal(msg []byte, payloadType byte, v interface{}) (err error) {
    // r := bytes.NewBuffer(msg)
    err = xml.Unmarshal(msg, v)
    return err
}

var XMLCodec = websocket.Codec{xmlMarshal, xmlUnmarshal}
```

We can then serialise Go objects such as a `Person` into an XML document and send it from a client to a server by

```
/* PersonClientXML
 */
package main

import (
    "golang.org/x/net/websocket"
    "fmt"
    "os"
    "xmlcodec"
)

type Person struct {
    Name   string
    Emails []string
}

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

    conn, err := websocket.Dial(service, "", "http://localhost")
    checkError(err)

    person := Person{Name: "Jan",
        Emails: []string{"ja@newmarch.name", "jan.newmarch@gmail.com"},
    }

    err = xmlcodec.XMLCodec.Send(conn, person)
    if err != nil {
        fmt.Println("Couldn't send msg " + err.Error())
    }
    os.Exit(0)
}

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

A server which receives this and just prints information to the console is

```
/* PersonServerXML
 */
package main

import (
    "golang.org/x/net/websocket"
    "fmt"
    "net/http"
    "os"
    "xmlcodec"
)

type Person struct {
    Name   string
    Emails []string
}
```

```
func ReceivePerson(ws *websocket.Conn) {
    var person Person
    err := xmlcodec.XMLCodec.Receive(ws, &person)
    if err != nil {
        fmt.Println("Can't receive")
    } else {

        fmt.Println("Name: " + person.Name)
        for _, e := range person.Emails {
            fmt.Println("An email: " + e)
        }
    }
}

func main() {

    http.Handle("/", websocket.Handler(ReceivePerson))
    err := http.ListenAndServe(":12345", nil)
    checkError(err)
}

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

Web sockets over TLS

A web socket can be built above a secure TLS socket. We discussed in Chapter 8: HTTP how to use a TLS socket using the certificates from Chapter 7: Security. That is used unchanged for web sockets. that is, we use `http.ListenAndServeTLS` instead of `http.ListenAndServe` .

Here is the echo server using TLS

```
/* EchoServer
 */
package main

import (
    "golang.org/x/net/websocket"
    "fmt"
    "net/http"
    "os"
)

func Echo(ws *websocket.Conn) {
    fmt.Println("Echoing")

    for n := 0; n < 10; n++ {
        msg := "Hello  " + string(n+48)
        fmt.Println("Sending to client: " + msg)
        err := websocket.Message.Send(ws, msg)
        if err != nil {
            fmt.Println("Can't send")
            break
        }

        var reply string
        err = websocket.Message.Receive(ws, &reply)
        if err != nil {
            fmt.Println("Can't receive")
            break
        }
        fmt.Println("Received back from client: " + reply)
    }
}

func main() {

    http.Handle("/", websocket.Handler(Echo))
    err := http.ListenAndServeTLS(":12345", "jan.newmarch.name.pem",
        "private.pem", nil)
    checkError(err)
}

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

The client is the same echo client as before. All that changes is the url, which uses the `"wss"` scheme instead of the `"ws"` scheme:

```
EchoClient wss://localhost:12345/
```

Conclusion

The web sockets standard is nearing completion and no major changes are anticipated. This will allow HTTP user agents and servers to set up bi-directional socket connections and should make certain interaction styles much easier. Go has nearly complete support for web sockets.