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:

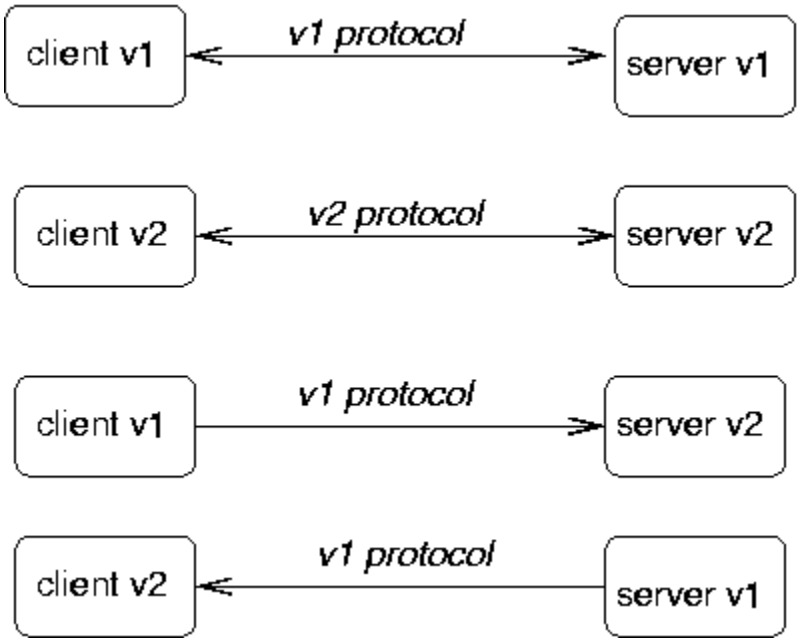
- Is it to be broadcast or point to point?
  Broadcast must be UDP, local multicast or the more experimental MBONE. Point to point could be either TCP or UDP.
- Is it to be stateful vs stateless?
  Is it reasonable for one side to maintain state about the other side? It is often simpler to do so, but 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 compatability 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 imposible.

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

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

- 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'.

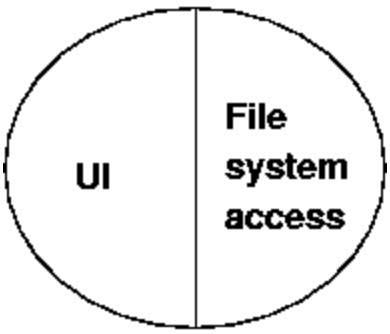
Simple Example

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.

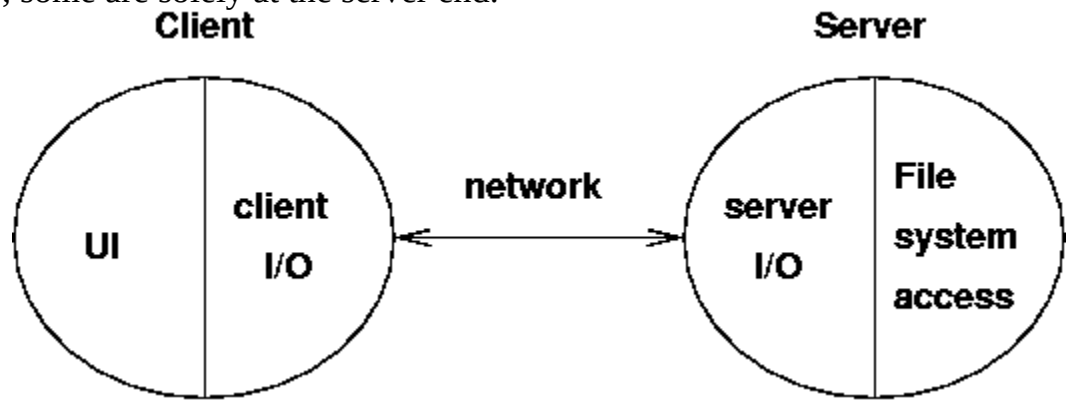
Look at a simple non-client-server program that allows you to list files in a directory and change and print the directory on the server. We omit copying files, as that adds to the length of the program without really introducing important concepts. For simplicity, all filenames will be assumed to be 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 italicised 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
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

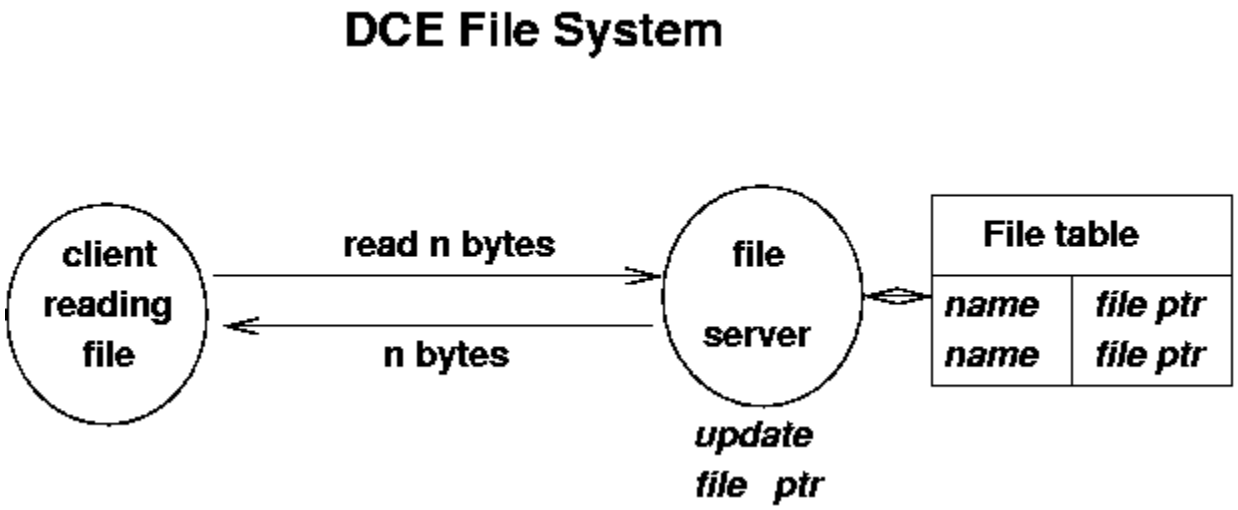
Applications often make use of state information to simplify what is going on. For example

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

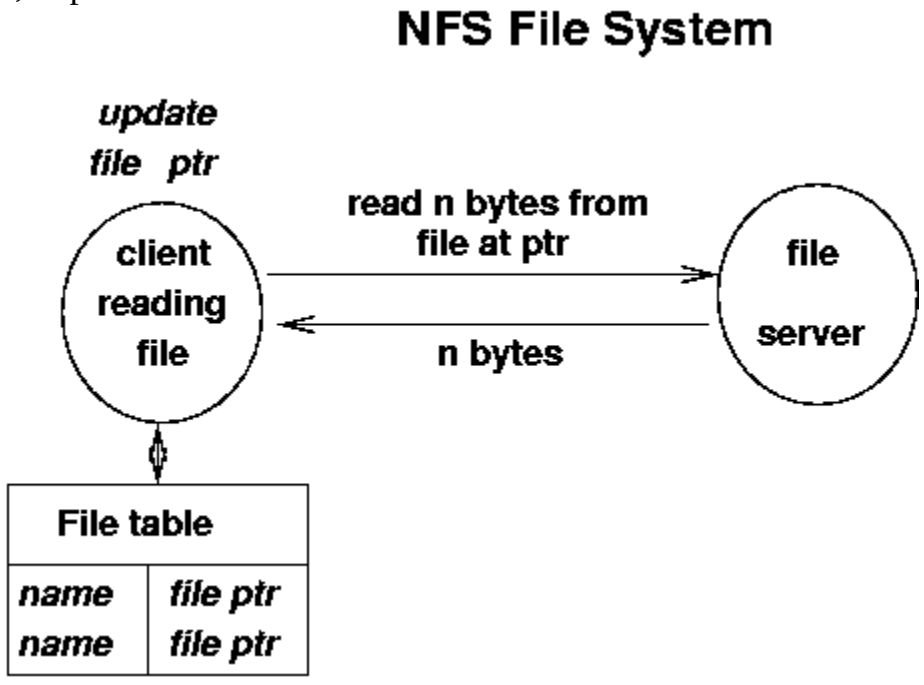
In a distributed system, such state information may be kept in the client, in the server, or in both.

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.



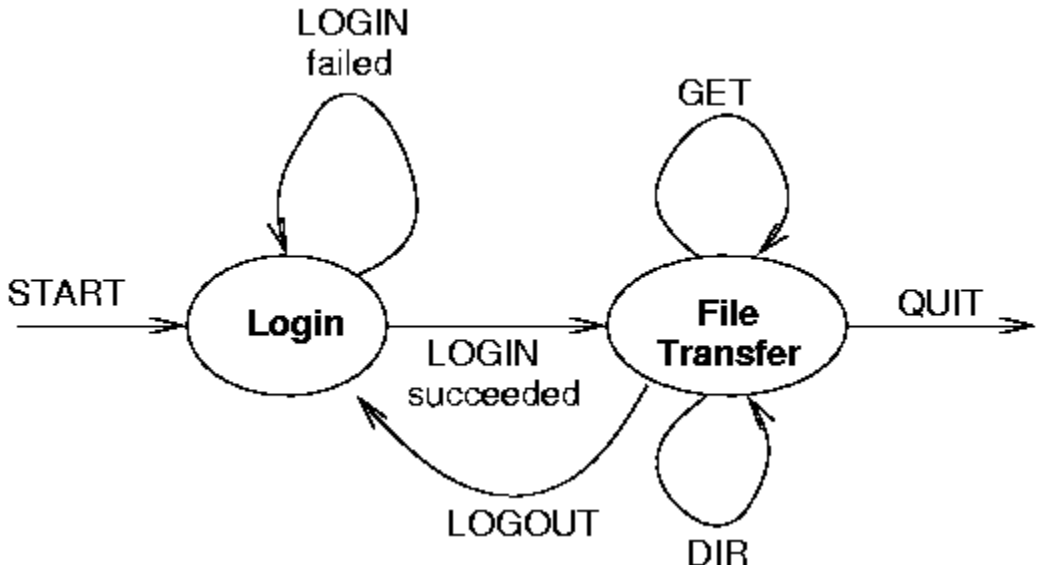
If the server maintains information about the client, then it must be able to recover if the client crashes. If information is not saved, then on each transaction the client must transfer sufficient information for the server to function.

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	Read	Write	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.

Summary

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.

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