Applications of Finite Automata

We have now devoted two chapters to the study of finite automata. This chapter will focus on applications of finite automata, and will include both simple and fairly advanced usages. Examples will be taken from the theory of operating systems, data communications, and network protocol analysis. We begin with a few very easy examples.

Goals for This Chapter

At first glance this chapter might seem a bit much for the student. A casual inspection will reveal that it covers a number of topics, including

1) Stages in a process,
2) Creation and persistence (saving to a file) of objects in a graphical system,
3) Process management in a time-sharing operating system,
4) Parity checking on a bit-serial communications line, and
5) Connection management for the Transmission Control Protocol.

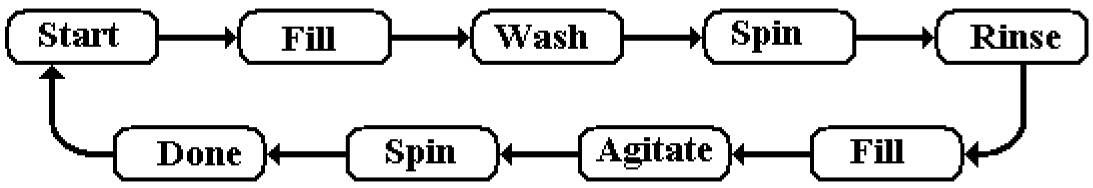
Each of these topics might easily cover a few weeks in a separate course; some such as management of TCP, would readily justify an entire course. The purpose of this chapter is not to teach these topics, but to show the utility of finite automata in many areas of study.

The author of these notes now faces a problem: how much to say about each application being discussed. The original draft of this chapter comprised only finite automata, labels of each state and transition, and very few comments. The idea behind this presentation was "Just look at the pictures and note how many uses they have". This approach was thought not to be sufficient, so the revised chapter adds some explanation when thought useful.

We begin this chapter of illustrations with the remark that finite automata can be found everywhere: traffic lights, vending machines, and washing machines (both dish washers and clothes washers). One of this author's friends, an MIT graduate, claimed to be able to model a baby as a finite state machine, but this is probably excessive zeal (or silliness) on his part.

Washing Machine

A basic washing machine is easily modeled as a finite automaton. Now that you understand Finite State Machines, you can get really clean clothes.



Progress Through the Computer Science Curriculum
It is not common to view a process in terms of a finite state machine, but it can be done.

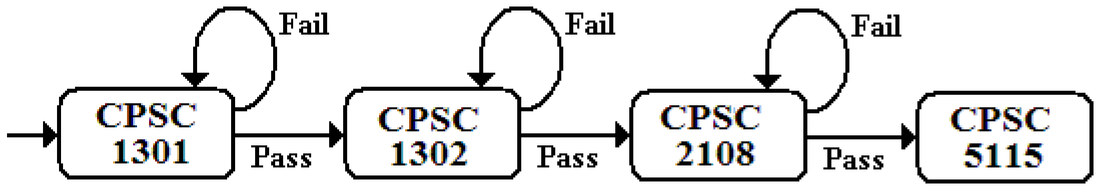
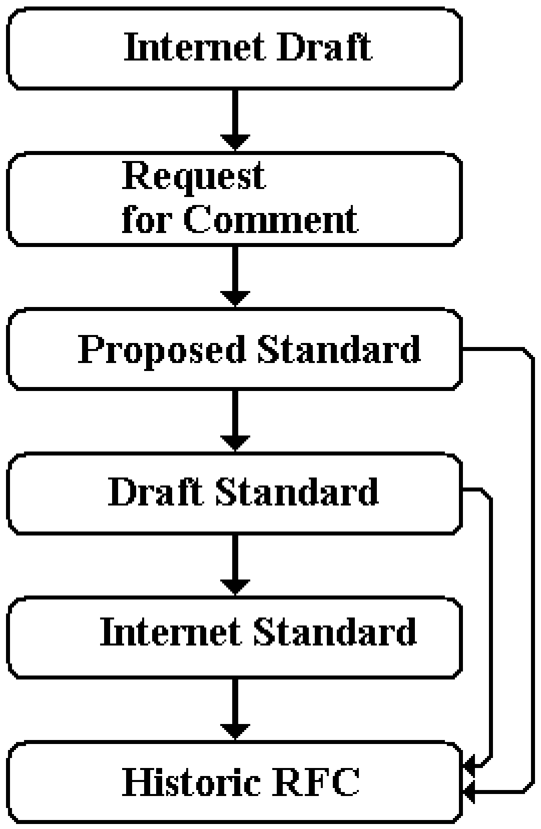


Figure: A Sequence of Courses in the CS Curriculum

One should note that finite automata are not often used to model processes such as this and the next one, for the reason that other models usually do it better. Nevertheless, we can view each of these two as true finite automata, although with fewer transitions than is usual.

Life Cycle of an Internet Standard
The managers of the Internet use published standards as a way to agree on rules for running the Internet. Each standard goes through a life cycle, beginning as an unofficial working copy and ending as a historic document. Here is a finite automaton modeling that process.



The **Internet draft** is a working copy. It is a work in progress with no official standing.

If an Internet draft shows sufficient promise, it is edited and published as a **RFC (Request for Comment).** At this stage other interested parties are invited to comment.

When the RFC becomes stable and is well understood, and is deemed to be of general interest, it becomes a **Proposed Standard**.

A proposed standard is elevated to a **Draft Standard** upon the completion of 2 successful independent and interoperable implementations.

Upon demonstration of more successful implementations the RFC becomes an **Internet Standard**.

Historic standards are either old Internet Standards that have been superceded or other drafts of sufficient interest to merit keeping.

Define a Line by Two Points

We now consider an example from object-oriented computer graphics. Recalling that two points specify a line, we consider a **line segment object** as being the line between any two points. There are many ways to specify an instance of a line segment object, we consider just one and show how a finite automaton model can be used in designing the code for the object.

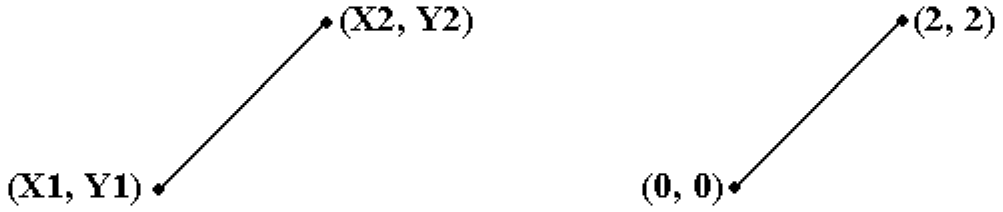
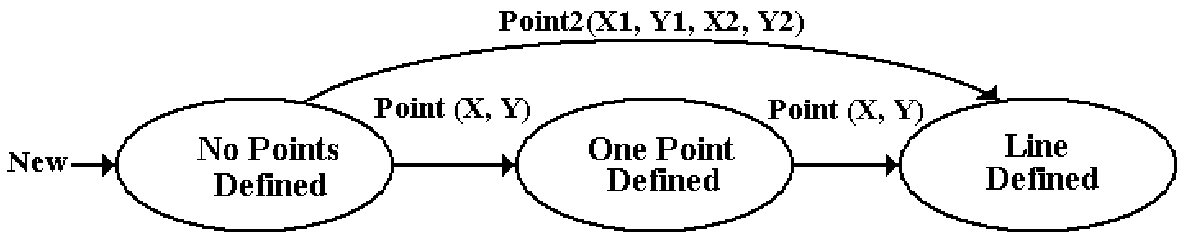


Figure: A Line Segment and An Example

The line segment is defined by two distinct end points and comprises all points on the line between those points. In our example, we have a line segment from point (0, 0) to point (2, 2), which can be modeled by the equation Y = X, with $0 \le X \le 2$. Note that the point (1, 1) is on the line segment. The point (6, 6) is not on the line segment, although it is on the line defined by the equation Y = X.

In this example, the code for the line segment object has three methods.

1. A **New** method to create an "empty" instance of the object;
   that is, with no data members defined.
2. A **Point(X, Y)** method that specifies one of the two points for the instance of the line segment object. One calls this method twice to specify the line segment, without making distinction between the two points.
3. A **Point2(X1, Y1, X2, Y2)** method that fully specifies the line segment instance.



The design requirement for the line segment object calls for it to be specified by two calls to the method Point(X, Y) or a single call to Point2(X1, Y1, X2, Y2); either

1) Point(0, 0) followed by a call to Point (2, 2), or
2) Point(2, 2) followed by a call to Point (0, 0), or
3) A single call to Point2(0, 0, 2, 2)

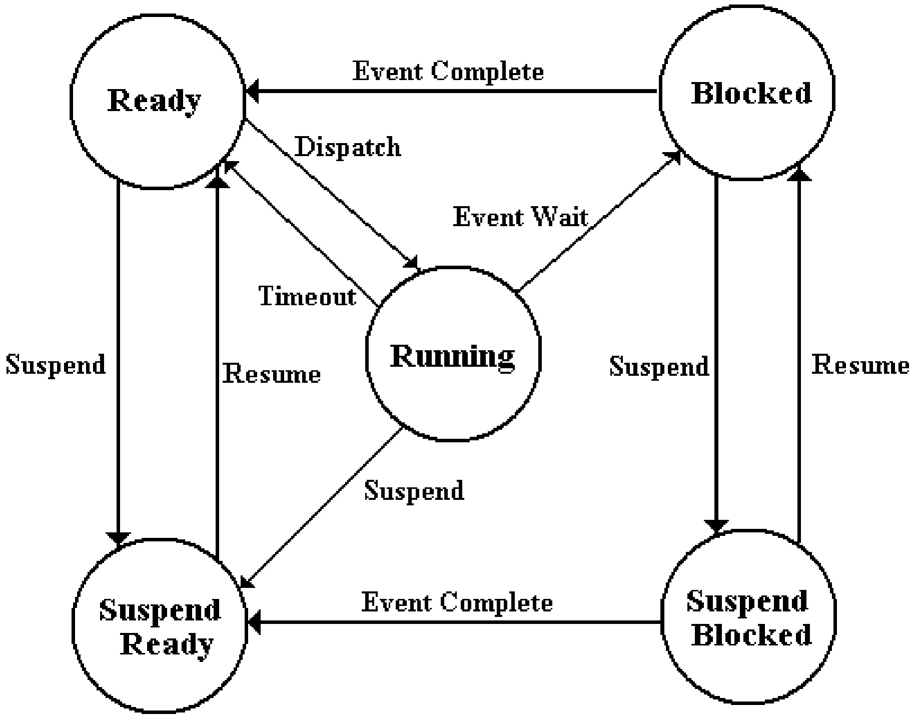
When the line segment has been specified completely, the instance of the object can be "persisted"; that is, stored in the database for the graphics system. The problem that this finite automaton model addresses is that of storing an incompletely specified line segment. If we create a new line and specify only one point by a single call to Point(X, Y) we do not have a complete description, and the line segment instance should not be persisted.

An Example from Operating Systems

We now consider process scheduling by an operating system running in time-sharing mode. In the theory of operating systems, the term "process" is so basic that it is impossible to define exactly, so we just think of it as a program that is running and using the CPU. There are three possible events that will cause a program to stop running, we consider only two.

1) The program terminates after finishing its computation. We ignore this.
2) The program requests I/O and must wait for the completion of the I/O, or the program must await the completion of an external event.
3) The program "times out". In time sharing mode, an operating system allocates to each program a set amount of time, called a "quantum", after which the OS stops running that program and allocates the CPU to another program that is ready to run.

Here is the finite automaton for the scheduling process.



If the process is **running** and exhausts its time allocation, it **times out** and is marked by the operating system as **ready** to run. The operating system maintains the list of programs ready to run in a queue (with possible priority considerations) and **dispatches** the next in line; that is allocates the CPU to the process and places it in the running state.

If a process is **running** and must wait on an event (I/O or other external event), it is placed in the **blocked** state. Upon completion of the external event, the process becomes ready to run and is moved to the ready state, from which it will eventually be dispatched.

Processes in either of the three principal states (running, ready, blocked) can be suspended. The two bottom states deal with suspended states, not of much interest to this author.

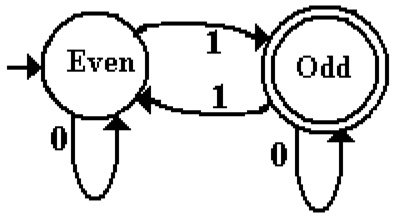
Parity Checker

We now consider the problem of checking the parity of a transmitted bit stream. The term "parity" generally applies to a binary bit stream (representing 0's and 1's) and reflects the number of 1's in the bit stream. The parity is odd if the number of 1's is odd and even if the number of 1's in the bit stream is even. Remember that 0 is an even number.

The principle behind the parity detector is the pair of simple rules:

Even_Number + 1 = Odd_Number
Odd_Number + 1 = Even_Number

We model the parity checker as a Moore machine, with an accept state indicating that the received string has the proper parity. The events causing transitions in the finite automaton are the receipt of a 0 (not changing the parity) and receipt of a 1 (changing the parity).



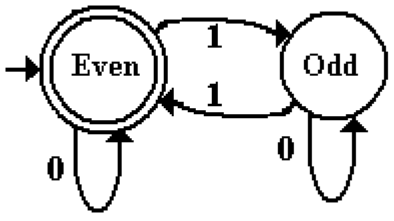


Figure: Odd Parity Detector Even Parity Detector

For those with experience in hardware design, this circuit can be implemented by a single T flip-flop with asynchronous preset to clear the flip-flop to 0 at the beginning of the string.

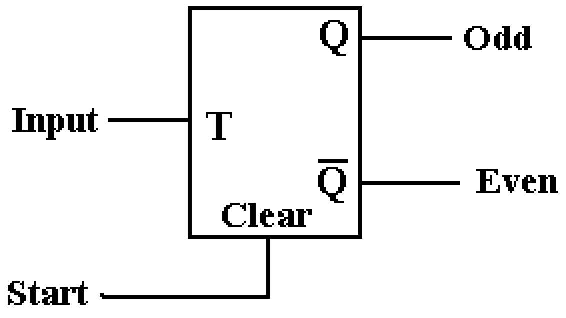


Figure: Single Flip-Flop Implementation of a Parity Detector

As an aside, we show how to extend the standard 7-bit ASCII character code to an 8-bit ASCII code with parity. This is only one of many options for data transmission.

Character	ASCII	7-bit code	8-bit code (even parity)	8-bit code (odd parity)
'A'	0x41	1000001	01000001	11000001
'C'	0x43	1000011	11000011	01000011

In this scheme the number of 1 bits in the 7-bit code is computed and a parity bit added (in our example placed as the leftmost bit) to make the parity correct.

Modeling the TCP

The TCP (Transmission Control Protocol) is a protocol used in connection with IP (Internet Protocol) – hence TCP/IP – to control interaction between a TCP Client and a TCP Server. More generically the two communicating TCP programs are called "TCP Hosts", but we shall consider the special case in which it is easier to speak of a "client" and a "server".

A **communications protocol** is a set of rules or conventions to be followed by two or more communicating entities. For example, there is the telephone protocol – the telephone rings and one answers "Hello" (this author's father always answered "Alright" – but he was a doctor and allowed a few eccentricities) to initiate the transfer. What we see in this telephone protocol is called a "handshake" in many areas of computer science. Two devices pass initial data in order to set up a communications link using an agreed protocol.

The conceptual model of communication via TCP is shown in the next figure. In general, the function of TCP is to allow a client application to communicate with a server application. The communication could be direct, in which case each application would be required to have a complete set of communication software installed. It is considerably easier to allow the TCP layer handle the communication and for the client and server to communicate only with TCP using the standard API (Application Program Interface).

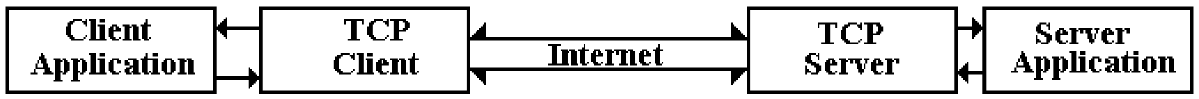


Figure: Applications Use TCP to Communicate

We now use a familiar application to reinforce the concept just mentioned. At the end of the semester the student will want to know his/her exam grade and final course grade. The most efficient way to do this is for the student to communicate via e-mail, but for this example we suppose that each student will call the "grade information server" (the instructor) using a cell phone. We are very comfortable with use of cell telephones to communicate, most of us having a long history using POTS (the land-line **P**lain **O**ld **T**elephone **S**ervice). We say things like "I'll call you" when we mean "I shall use my telephone and cause it to call your telephone, after which we can communicate using our two telephones since they will have established a reliable communications connection". Neither of us, using a cell phone, will worry about selection of radio frequency to communicate with the cell tower, nor will we worry about the process of establishing each of the two links to the cell tower that are required for the communication to be successful. It is the same with two communicating processes. The Client Application communicates directly with the Client TCP, the Server Application communicates directly with the Server TCP, and the two TCP processes manage the communication link without intervention of the applications.

We might as well mention a few issues with the nomenclature. The first is that the term TCP is used not only for the protocol itself but also for the software that implements the protocol. The second issue is that we often hear the term "TCP Protocol" which is translated literally as "Transmission Control Protocol Protocol" - a redundancy. But then again, we often use the term "Pizza Pie" which literally means "Cheese Pie Pie". Go figure.

TCP is a **connection-oriented protocol**, by which we mean that the TCP client and the TCP server set up a reliable connection between the client application and the server application. The connection is managed by what is called a **handshaking procedure**: the setup by a three-way handshake and the tear-down by a four-way handshake.

The **three-way handshake** between the TCP Client and TCP Server begins with the Server Application which tells the TCP Server that it is ready to accept a connection. It does this by issuing what is called a "**passive open**". It now awaits communication from the TCP Client. The handshake proceeds as follows.

1) The Client Application issues an "**active open**" to the Client TCP. The Client TCP then sends a packet to the Server TCP to announce its wish for a communication and includes information about its ability to communicate.
2) The Server TCP responds with two packets.
   a) An acknowledgement of the request of the Client Server, and
   b) A packet containing information about its ability to communicate.
3) The Client TCP responds with a packet acknowledging the TCP Server packet.

The **four-way handshake** reflects the fact that the Client TCP will end its transmission to the Server TCP before the Server TCP drops its communication with the TCP Client – the Client stops asking for information a bit before the Server stops sending it.

1) The Client Application issues an "active close" to the Client TCP, which sends a FIN packet to the Server TCP.
2) The Server TCP acknowledges the request to close by sending an ACK packet to the Client TCP. The Server TCP continues to transmit data until it is finished.
3) When the Server TCP has no more data to send, it sends a FIN packet to the Client.
4) The Client TCP sends an ACK to the Server TCP to acknowledge the FIN.

Note that the finite automata shown below use the "Input/Output" labeling of the transitions. The following examples will illustrate the notation.

Passive Open / --
When the TCP Server is in the Closed state and receives a Passive Open, it moves to the Listen state without transmitting anything to the (as yet not existent) Client TCP.

Active Open / SYN
When the TCP Client is in the Closed state and receives an Active Open, it transmits a SYN packet to the TCP Server and moves to the SYN-SENT state.

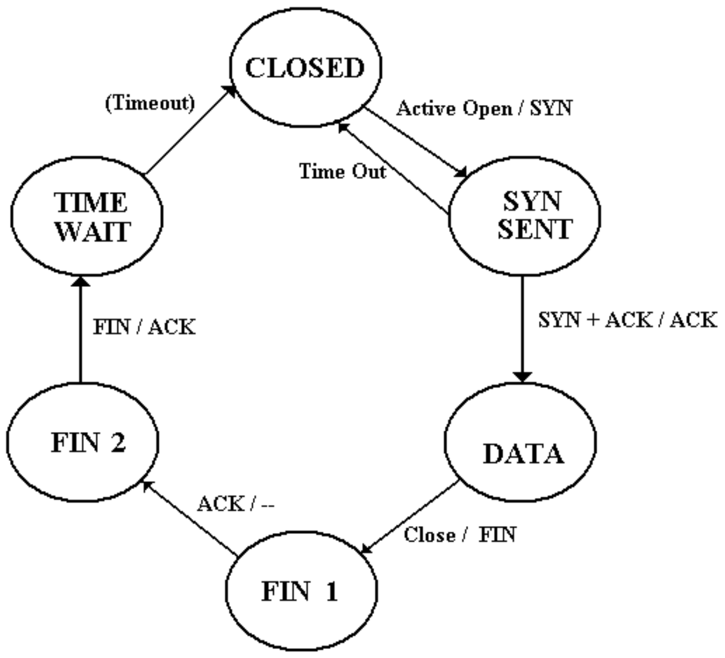
SYN / SYN + ACK
This probably should be labeled SYN / (SYN + ACK) to show that the TCP Server receives a SYN and replies with a packet containing both SYN and ACK.

SYN + ACK / SYN
This probably should be labeled (SYN + ACK) / SYN as it shows what the TCP Client does when receiving a packet containing both SYN and ACK.

The reader should note that neither the TCP Client finite automaton nor the TCP Server finite automaton is complete, as neither have transitions to handle error conditions. We repeat the earlier disclaimer – this is not a course in TCP, just an illustration of finite automata.

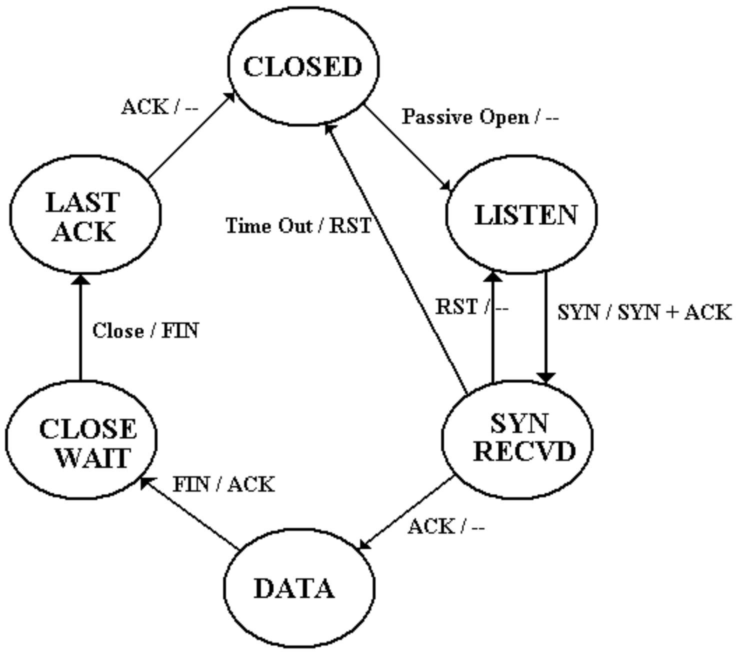
TCP Client Diagram

Here is the finite automaton representing the TCP Client. Remember that the TCP client is that part of the TCP program that connects the client software to the Internet, handling the communications via the Transmission Control Protocol.



1. The Client TCP starts in the CLOSED state.
2. In the CLOSED state, the Client receives an Active Open request from the client application program and sends a SYN signal to the Server TCP and goes to SYN SENT.
3. In SYN SENT, the Client either times out and returns to closed or receives SYN + ACK, a combined signal from the Server TCP and goes to the DATA transfer state.
4. The Client remains in the DATA transfer state until it receives a Close request from the client application, in which case it sends a FIN to the Server TCP and goes to FIN 1.
5. In the FIN 1 state, the Client TCP awaits the ACK from the Server TCP. When it receives the ACK it goes into the FIN 2 state.
6. In the FIN 2 state the Client TCP waits for the TCP server to close the connection, which it does by sending a FIN. When Client TCP receives the FIN, it sends an ACK to the Server TCP and goes to the TIME WAIT state.
7. In the TIME WAIT state, the Client TCP sets a timer and waits for it to time out. It does this at a value sufficient to allow duplicate IP packets to arrive at their destination. When the timer expires, the Client TCP goes to the CLOSED state.

TCP Server Diagram



1. The Server TCP starts in the closed state.
2. The server application sends a Passive Open to the Server TCP, which then goes into the LISTEN state.
3. While in the LISTEN state, the Server TCP can receive a SYN signal from the Client TCP. It responds by sending a SYN + ACK combo and going to SYN RECVD.
4. In the SYN RECVD state, the Server TCP receives an ACK from the Client TCP and moves into the DATA transfer state.
5. When the Client TCP wants to close the connection, it sends a FIN to the Server TCP, at which time the Server TCP replies with an ACK and moves to CLOSE WAIT.
6. When the Server TCP is in the CLOSE WAIT state, it waits until it receives a Close request from the server application, at which time it sends a FIN to the Client TCP and goes to the LAST ACK state.
7. In the LAST ACK state, the Server TCP is awaiting the last ACK from the Client TCP. On receiving this, it goes into the CLOSED state.

Reference: All material on TCP was taken from the book
TCP/IP Protocol Suite by Behrouz A. Forouzan, published by McGraw-Hill in 2000.
ISBN 0 – 256 – 24166 – X