

**SOFTWARE IMPLEMENTATION
OF
TCP & IP
PROJECT REPORT**

Submitted in partial fulfillment of the requirements for the award of the degree
of

M.Sc Applied Science (Software Engineering)
Bharathiar University,
Coimbatore.

P-1123

Submitted by

SENTHIL KUMAR. G
Reg.No. 9937S0092

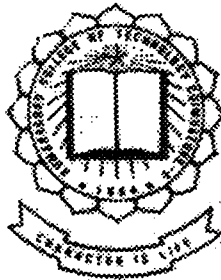
Guided By

External Guide

Internal Guide

Mr. Meenakshi Sundaram, B.E.

Mr. P.K.Jayaprakash, M.C.A.



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
KUMARAGURU COLLEGE OF TECHNOLOGY
COIMBATORE - 641 006

CERTIFICATE

This is to certify that this project work entitled

“ SOFTWARE IMPLEMENTATION OF TCP & IP ”

Submitted to

KUMARAGURU COLLEGE OF TECHNOLOGY

In partial fulfillment of the requirements for the award of the degree

Of

M.Sc. APPLIED SCIENCE (Software Engineering)

The record work done by

SENTHIL KUMAR. G

Reg.No. 9937S0092


During his period of study in the Department of Computer Science and Engineering, Kumaraguru College of Technology, Coimbatore - 641 006, under my supervision and guidance and this project work has not formed the basis for the award of any guidance and this project work has not formed the basis for the award of any degree/ Diploma/ Associate ship/ Followed or similar title to any candidate of any university.

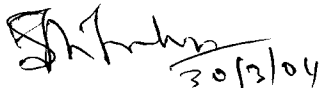

Professor and Head


Staff-in-charge

Submitted for University Examinations held on

.....30.3.2004.....


Internal Examiner


30/3/04
External Examiner

DECLARATION

I hereby declare that the project work entitled

“ SOFTWARE IMPLEMENTATION OF TCP & IP ”

Done at

HCL TECHNOLOGIES

Submitted to

KUMARAGURU COLLEGE OF TECHNOLOGY

In partial fulfillment of the requirements for the award of the degree

M.Sc. APPLIED SCIENCE (Software Engineering)

Is a report of work done by me during my period of study in Kumaraguru
College of Technology, Coimbatore - 641 006

Under the supervision of

Mr. K.R.Baskaran

**Assistant Professor, Dept of Computer science & Engineering,
Kumaraguru College of Technology, Coimbatore.**

Place : Coimbatore

Date :

Signature of the Candidate

(Senthil Kumar. G)

Staff-in-charge

**Mr. K.R.Baskaran,
Assistant Professor, Dept of Computer Science & Engineering,
Kumaraguru College of Technology, Coimbatore.**

HCL TECHNOLOGIES LTD
Networking Products Division
No.49-50, Nelson Manickam Road
Chennai - 600 029, India
Tel: +91-44-23741939
Fax: +91-44-23741038
Website: www.hcltech.com

HCL TECHNOLOGIES

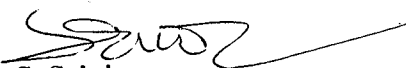
8 March 2004

TO WHOMSOEVER IT MAY CONCERN

This is to certify that **Mr. Senthil Kumar. G**, M.Sc (Software Engineering), student of **Kumaraguru College of Technology** was involved in a project in our organization for two months from **December 2003** to **March 2004**. He was assigned to project "**Software implementation of TCP & IP**" under our guidance. His performance in the above mentioned project was good.

Yours Truly,

For HCL Technologies.



S. Srinivasan
Project Manager.

ACKNOWLEDGEMENT

To add meaning to the perception, it is my indebtedness to honor a few who had helped me in this endeavor, by placing them on record.

With profound gratitude, I am extremely thankful to **Dr.K.K.Padmanaban B.Sc. (Engg), M.Tech, Ph.D.**, Principal, Kumaraguru Collage of Technology, coimbatore for providing me an opportunity to undergo the M.Sc [APPLIED SCIENCE] SOFTWARE ENGINEERING course and thereby this project work also.

I extend my heartfelt thanks to my Computer Science & Engineering Department head **Prof.Dr.S.Thangasamy B.E (Hons), Ph.D.**, for his kind advice and encouragement to complete this project successfully.

It's my privilege to express my deep sense of gratitude and profound thanks to **Mr.Srinivasan, Senior Software Developer**, HCL Technologies Ltd, Chennai for having allowed me to do my project work in his esteemed team and for helping me in all means in successful completion of this project work.

Gratitude will find least meaning without thanking my course coordinator **Mr.K.R.Baskaran** B.E., M.S Assistant Professor, Computer Science & Engineering department and guide **Mr.P.K.Jayaprakash** M.C.A., Lecturer, Dept of Computer science & Engineering, for the valuable guidance and support throughout my project.

Words are boundless for me to express my deep sense of gratitude and profound thanks to **Mr. Meenakshi Sundaram** and all my associates at HCL, for all their kind guidance and encouragement towards my project work.

My gratitude is due to all staff members of CSE department, my parents and all my friends for their moral support and encouragement for successful completion of my project.

G.Senthil Kumar

PROJECT ABSTRACT

Despite the widespread use and popularity, the details of TCP/IP protocols, the structure of the software that implements them & their interaction remain a mystery to most computer professionals. To illustrate all these subtleties and internal details, we designed and built a working system that implements the key protocols in the suite: TCP, UDP, IP & ARP.

The code attempts to conform to the protocol standards and to include current ideas. For example, our TCP code includes silly window avoidance and the Jacobson-Karels slow-start and congestion avoidance optimizations, features sometimes missing from commercial implementations. However we are realistic enough to realize that the commercial world does not always follow the published standards, and have tried to adapt the system for use in practical environment.

We have implemented the system under the Xinu Operating System. Xinu is a small, elegant operating system that has many features similar to UNIX. Several vendors have used versions of Xinu as an embedded OS in commercial products.

The documentation explores the design choices that underlie our software, and describes the data structures and procedures that implement the protocols. It focuses on our implementation, including the source code that forms the working system. Although this implementation of ours was not developed as a commercial product, it obeys protocol standards.

We don't claim that the code we had developed is completely bug-free, or even that it is better than other implementations. Indeed, after using it, we continue to find ways to improve our software.

CONTENTS

1. INTRODUCTION	
1.1 Organization Profile.....	1
1.2 Problem Definition	2
2. SYSTEM ANALYSIS	
2.1 System Requirements.....	7
2.2 Scope of the System	11
2.3 System Environment	11
3. SYSTEM DESIGN	
3.1 Module Design	
3.1.1 Address Resolution Protocol	13
3.1.2 The Internet Protocol.....	15
3.1.3 The User Datagram Protocol.....	16
3.1.4 The Transmission Control Protocol	17
4. SYSTEM IMPLEMENTATION	
4.1 Testing and Test Plan.....	23
4.2 Testing Methods.....	24
5. LIMITATIONS AND FUTURE ENHANCEMENTS.....	26
6. CONCLUSION	28
7. REFERENCES	30
8. APPENDIX A – Sample Code	32

Introduction

1. INTRODUCTION

1.1 ORGANIZATION PROFILE

HCL Technologies is one of India's leading global IT services and product engineering companies, providing value-added, software-led IT solutions and services to large- and medium-scale organizations.

At HCL Technologies, we've cut our teeth on technology and that's what we understand best. We don't stop asking ourselves some basic questions: Is there a better solution? A different approach? A new technology? A more viable process? A more cost-effective proposition?

The testimony of our technological prowess lies in the dedicated Offshore Development Centers we operate for some of the world's leading organizations. Our clientele includes over 370 prestigious organizations in the world, including 40 Fortune 500 companies.

Our presence across 14 countries gives us global reach and a vast rollout support capability. Together with our formidable team of high-caliber software professionals we have successfully positioned ourselves at the vanguard of the global IT services revolution.

We continue to reiterate our key imperatives of focusing on our technology competencies and keeping pace with continuous growth and learning. We believe that in technology there are no half-measures.

Our growth has been a result of our unique business model and clearly defined growth strategies. Offshore-led, technology-centric, powered by domain expertise and a comprehensive understanding of diverse business verticals, HCL Technologies' growth has been balanced and keeps exploring new dimensions.

1.2 PROBLEM DEFINITION

The TCP/IP software is part of the computer operating system. It uses a separate thread of execution to isolate pieces of protocol software, making it much easier to design, understand, and modify. Each thread or process executes independently, providing apparent parallelism. Like most protocol software, our implementation has a separate process for IP, TCP input, TCP output & TCP timer management.

We have implemented the protocols under the Xinu operating system. Xinu provides mechanisms that processes can use to synchronize their execution. We use counting semaphores for mutual exclusion (i.e., to guarantee that only one process accesses a piece of code at a given time), and for producer-consumer relationships (i.e., when a set of processes produces a set of data items that another set of processes consume). We also use a port mechanism that allows processes to send messages to one another using a finite queue. The port mechanism uses semaphores to coordinate the processes that use a finite queue. If a process attempts to send a message to a port that is already full, it will be blocked until another process extracts it. Similarly, if a process attempts to extract a message from an empty port, it will be blocked until another process deposits a message in the port.

Processes implementing protocols use both conventional queues and ports to pass packets among themselves. For example, the IP input process sends TCP segments to a port from which TCP extracts, while network input processes use a queue to deposit arriving datagrams to the IP. When data is passed through conventional queues, the system must use message passing or semaphores to synchronize the actions of independent processes.

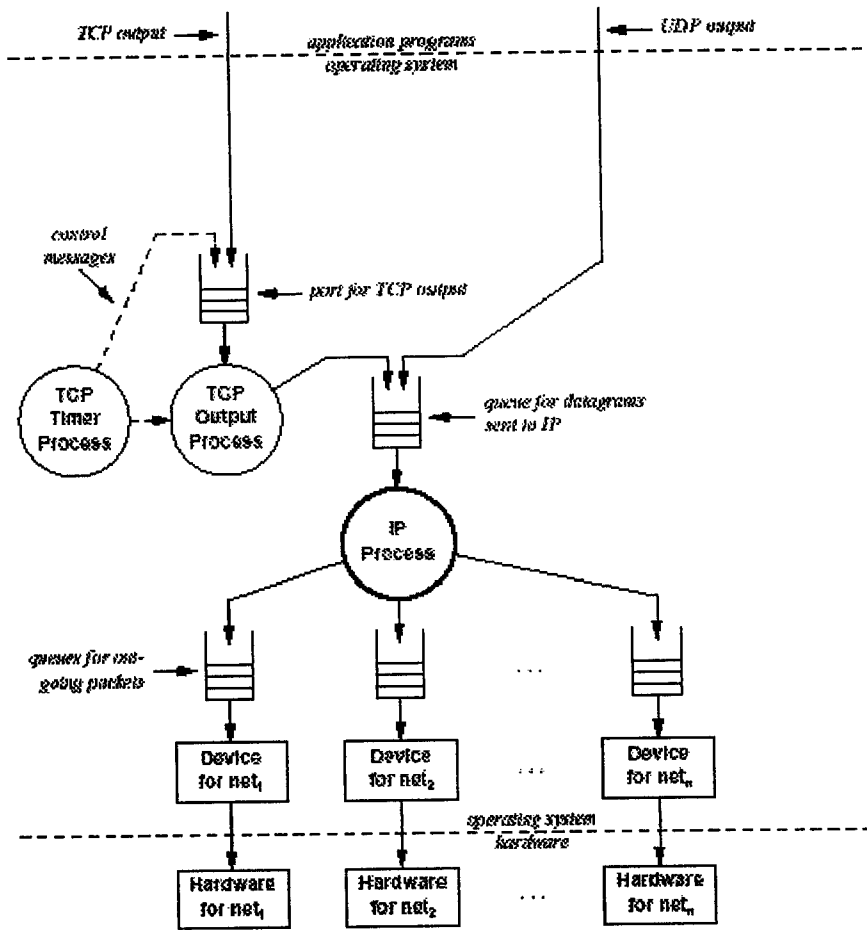
OUTPUT PROCESS STRUCTURE:

An application program, executing as a separate process, calls system routines to pass stream data to TCP or datagrams to UDP. For UDP output, the process executing the application program transfers into operating system (through a system call), where it executes UDP procedures that allocate an IP datagram, fill in the appropriate destination address, encapsulate the UDP datagram in it, and send the IP datagram to the IP process for delivery.

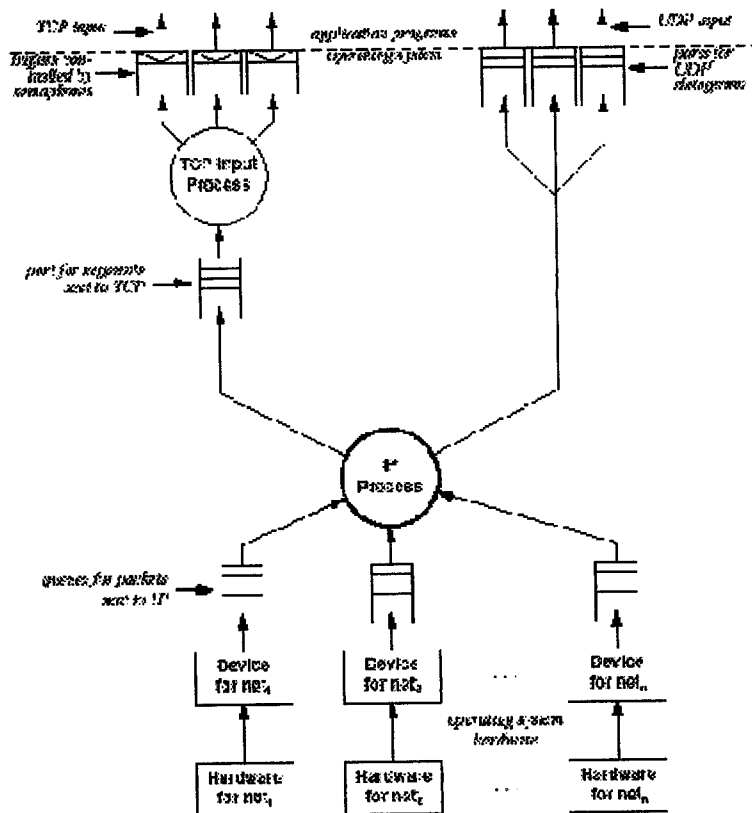
For TCP output, the process exciting an application program calls a system routine to transfer data across the operating system boundary and place it in a buffer. The application process then informs the TCP output process that new data is waiting to be sent. When TCP output process executes, it divides data streams into segments and encapsulates each segment in an IP datagram for delivery. Finally, the TCP output process enqueues the IP datagram on the port where IP will extract & send it.

INPUT PROCESS STRUCTURE:

The network device drivers enqueues all incoming packets that carry IP datagrams on queues for the IP process. IP extracts incoming packets from the queues and demultiplexes them, delivering each packet to the appropriate higher-level protocol software. When IP finds a datagram carrying UDP, it invokes UDP procedures that deposit the incoming datagram on the appropriate port, from which the application program reads it. When IP finds a datagram carrying a TCP segment, it passes the datagram to a port from which the TCP input process extracts it.



Output process structure showing the path of data between an application program and the network hardware. Output from the device queues is started at interrupt time. IP is a central part of the design the software for input and output both share a single IP process.



Input process structure showing the path of data between the network hardware and an application program. Input to the device queues occurs asynchronously with processing. IP is a central part of the design; the software for input and output share a single IP process.

System Analysis

2. SYSTEM ANALYSIS

2.1 SYSTEM REQUIREMENTS

2.1.1 HANDLING NETWORK INPUT & INTERRUPTS

To accommodate random packet arrivals, the system needs the ability to read packets from any network interface. It is possible to solve the problem of waiting for a random interface in several ways. Some Operating system's use the computer's software interrupt mechanism. When a packet arrives, a hardware interrupt occurs and the device driver performs it's usual duties of accepting the packet, and restarting the device. Before returning from the interrupt, the device driver tells the hardware to schedule a second, lower priority interrupt. As soon as the hardware completes, the lower priority interrupt occurs as though another hardware interrupt occurred. This second interrupt is known as software interrupt, suspends processing and causes the CPU to jump to code that will handle it.

Software interrupts are efficient, but require hardware that are not available in all computers. To make our protocol software portable, we chose to avoid software interrupts and designed code that relies only on the conventional hardware interrupt.

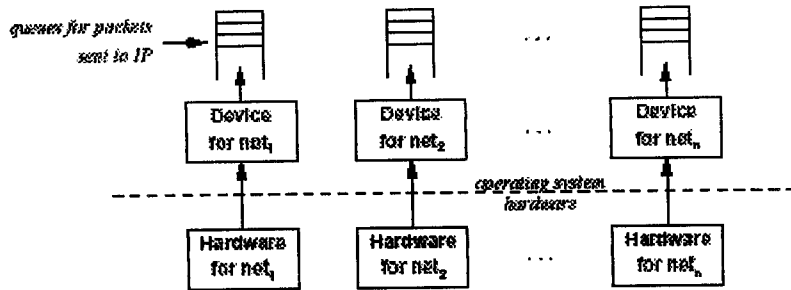


Figure 2.1 The flow of packets from the network interface hardware through the device driver in the operating system to an input queue associated with the device.

2.1.2 PASSING PACKETS TO HIGHER LEVEL PROTOCOLS

Communication between the network device drivers and the process that implements IP uses a set of queues. When a data gram arrives, the network input process enqueues it and sends message to the IP process.

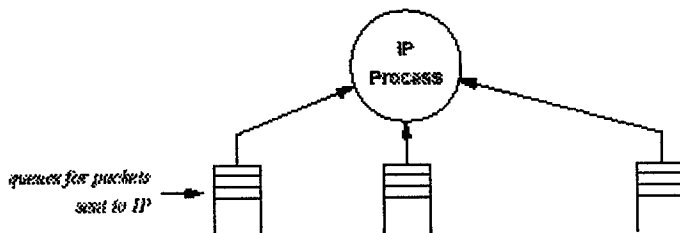


Figure 2.2 Communication between the network device drivers and the process that implements IP uses a set of queues. When a datagram arrives, the network input process enqueues it and sends a message to the IP process.

2.1.3 PASSING DATAGRAMS FROM IP TO TCP

Because TCP is complex, we use a separate process to handle incoming segments. A consequence of using separate process for IP & TCP processes is that they must use interprocess communication mechanism when they interact. Our implementation uses the port mechanism. IP calls *psend* to deposit segments in a port, and TCP calls *perceive* to retrieve them.

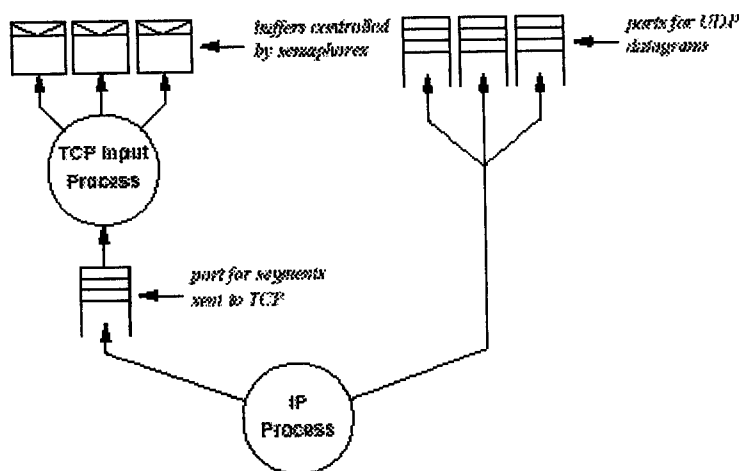


Figure 2.3 The flow of datagrams through higher layers of software. The IP process sends incoming segments to the TCP process, but places incoming UDP datagrams directly in separate ports where they can be accessed by application programs.

2.1.4 PASSING DATAGRAMS FROM IP TO UDP

IP process executes the UDP procedures that handle an incoming datagram. These procedure examine the destination UDP protocol number, use it select an operating system queue for the incoming datagram. The IP process deposits the UDP datagram on the appropriate port, where an application program can extract it.

2.1.5 FROM TCP THROUGH IP TO NETWORK OUTPUT

Like TCP input, TCP output is also complex. Connections must be established, data must be placed in segments, and the segments must be retransmitted until the acknowledgements arrive. Once the segment has been placed in a datagram, it can be passed to IP for routing & delivery. We use two TCP processes to handle the complexity.

The first called *tcpout*, handles most of the segmentation and data transmission details. The second called *tcptimer*, manages the timer, schedules retransmission timeouts, and prompts *tcpout* when a segment must be retransmitted.

The *tcpout* process uses a port to synchronize input from multiple processes. TCP is stream oriented meaning that it allows an application to send few bytes of data at a time. Consequently items in the port do not correspond to individual segments. Instead a process that emits data, places the data in a output buffer and places a message in the port informing TCP that it has sent data. The *timer* process deposits a message in the port whenever a timer expires and TCP has to retransmit the segment. Thus the port can be thought of a queue of events that TCP must handle.

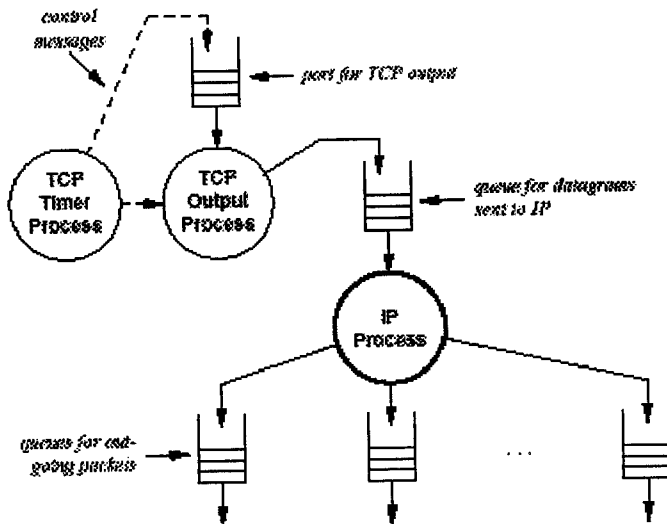


Figure 2.5 The TCP output and timer processes use the IP process to send data.

2.2 SCOPE OF THE SYSTEM

The scope of the system is to provide active experimentation with the TCP/IP suite of protocols. Our soul purpose is to develop source code for the protocols & make them interact using the inter process communication primitives.

2.3 SYSTEM ENVIRONMENT

Our implementation of TCP/IP has been developed using C on Xinu platform. Xinu is a small, elegant operating system that has features similar to UNIX. Several vendors have used versions of Xinu as an embedded system in commercial products.

System Design

3. SYSTEM DESIGN

3.1 MODULE DESIGN

3.1.1 THE ADDRESS RESOLUTION PROTOCOL

ARP binds high-level IP addresses to low level physical addresses. Address binding software forms the boundary between higher layers of protocol software, which uses only IP addresses, and lower layers of device driver software, which use only hardware addresses.

Conceptually, we have divided the ARP implementation of ours into three modules: *an output module, input module, and a cache manager.*

- **The ARP Output Module:** When sending a datagram, the network interface software calls a procedure in the output module to bind an IP address to an Hardware address. The output procedure returns the binding, which the network interface routines use to encapsulate & transfer the packet.

- **The ARP Input Module:** The Input module handles ARP packets that come from the network; it updates the ARP cache by adding new bindings.

- **The ARP Cache Manager:** The Cache Manager implements the Cache Replacement policy; it examines entries in the cache and removes them when they reach a specified age.

For implementing the Address Resolution Protocol, we have followed certain simple *design rules*:

- **Single Cache:** We have used a single physical cache that holds value for all the networks. Each field entry in the cache field contains which specifies the network from which the binding was obtained.
- **Global Replacement Policy:** When replacing the cache entries with another entry, we don't check if the old and the new entry correspond to the same network. Any existing item in the cache can be removed, independent of whether the new binding comes from the same network or not.
- **Cache Timeout & Removal:** We use a time-to-live field for each entry in the ARP cache for handling this issue. Whenever an entry is added to the cache, we initialize the time-to-live field on the entry. As time proceeds, the cache manager decrements this value, and discards the entry when the value reaches zero. This removal is independent of the frequency with which the entry was used.
- **Multiple Queues of Waiting Packets:** If many packets are destined for the same IP address & the required binding is not in the Cache, the packets are queued until the binding arrives for that address. Like this, there happens to be a queue of outgoing packets for each entry in the ARP cache.



- **Exclusive Access to the Cache:** The procedure that searches the ARP cache, require exclusive access. But the searching procedure does not have code for mutual exclusion. Responsibility to ensure mutual exclusion falls to the caller.

3.1.2 THE INTERNET PROTOCOL

Conceptually, the IP is a central switching point in the protocol software. It accepts incoming datagrams from the network interface software as well as outgoing datagrams that higher-level protocols generate. After routing a datagram, IP either sends it to one of the network interfaces or to a higher-level protocol on the local machine.

Our IP implementation has two distinct parts: one that handles input and one that handles output. The input part uses the PROTO field of the IP header to decide which higher-level protocol module should receive an incoming datagram. The output part uses the local routing table to choose a next hop for outgoing datagrams.

In addition IP software must work in gateways. In particular gateway software cannot be easily partitioned into input & output because a gateway must forward an arriving datagram on to its next hop. Thus IP might also generate output while handling an input datagram.

Our implementation of IP uses three main *design primitives*:

- **Uniform Input Queue & Uniform Routing:** We have decided to use a single input queue style for all datagrams IP must handle, independent of whether they arrive from the network or local machine. In our code, we do not use a special case for locally generated datagrams. All datagrams are treated as same & a single routing algorithm is used to route all of them.
- **Independent IP Process:** Our IP software executes as a single, self-contained thread of control. Using a process for IP keeps the software easy to understand and modify.
- **Local Host Interface:** To avoid making delivery to the local machine a special case, we use a pseudo-network interface for local delivery. When the IP algorithm routes each datagram, it sends the datagram to this local interface. The Local Interface uses the PROTO field to determine which protocol software on the local machine to receive the datagram.

3.1.3 THE USER DATAGRAM PROTOCOL

Conceptually, communication with UDP is quite simple. The protocol standard specifies an abstraction known as the *protocol port number* that application programs use to identify the endpoint of communication. When an application program on machine A wants to communicate with an application on machine B, each application must obtain a UDP port number from its local operating system. Both applications should use these port numbers when they communicate.

UDP provides both pair wise communication between peer programs and many-one communication between clients and server. While the two basic styles of demultiplexing both support clients and servers, each has advantages and disadvantages. We choose to demultiplex using only the destination protocol number, and make creation of server trivial. To help support clients, the system includes a procedure that generates a unique (unused) protocol port number on demand.

Our implementation can be split into UDP input, UDP output & UDP checksum. The IP process executes the UDP input procedure, which demultiplexes datagrams and deposits each on a queue associated with the destination protocol port. Application programs allocate a port for transmission and then call the output procedures to create and send UDP datagrams. The UDP checksum is used to verify if the IP datagram that carries the UDP, contains the correct IP source and destination address.

3.1.4 THE TRANSMISSION CONTROL PROTOCOL

Our implementation of TCP uses three processes. One Process handles incoming segments, another manages outgoing segments, and the third is a timer that manages delayed events such as transmission timeout. In theory, using separate processes isolates the input, output, and event timing parts of TCP and permits us to design each piece independently. In practice, however the processes interact closely. For example, the input & output process must co-operate to match incoming acknowledgements with outgoing segments and cancel the corresponding timer retransmission event. Similarly, the output & the timer process interact when the output process schedules a retransmission event or when the timer triggers a retransmission.

Transmission Control Blocks:

TCP uses a data structure called TCB for every active connection, which contains all information about the connection, including the addresses and port numbers of the connection end points, the current round-trip time estimate, data that has been sent or received, whether acknowledgement or retransmission is needed or not, and any statistics TCP gathers about the use of the connection.

TCP Input Processing and the TCP Finite State Machine:

Our implementation uses a procedure driven implementation of the TCP finite state machine in which one procedure correspond to each state. Conceptually, TCP uses this finite state machine to control all interactions. Each end of a TCP connection implements a copy of the TCP state machine and uses it to control actions taken when a segment arrives. The TCP state machine completely specifies how TCP on one machine interacts with TCP on another.

In practice, however the state machine does not fully specify interactions. Instead, the machine specifies only the macroscopic state of TCP, while additional variables specify the details or microscopic state. More important, because the macroscopic transmission specifies the state machine do not control output or retransmission, such events must be handled separately.

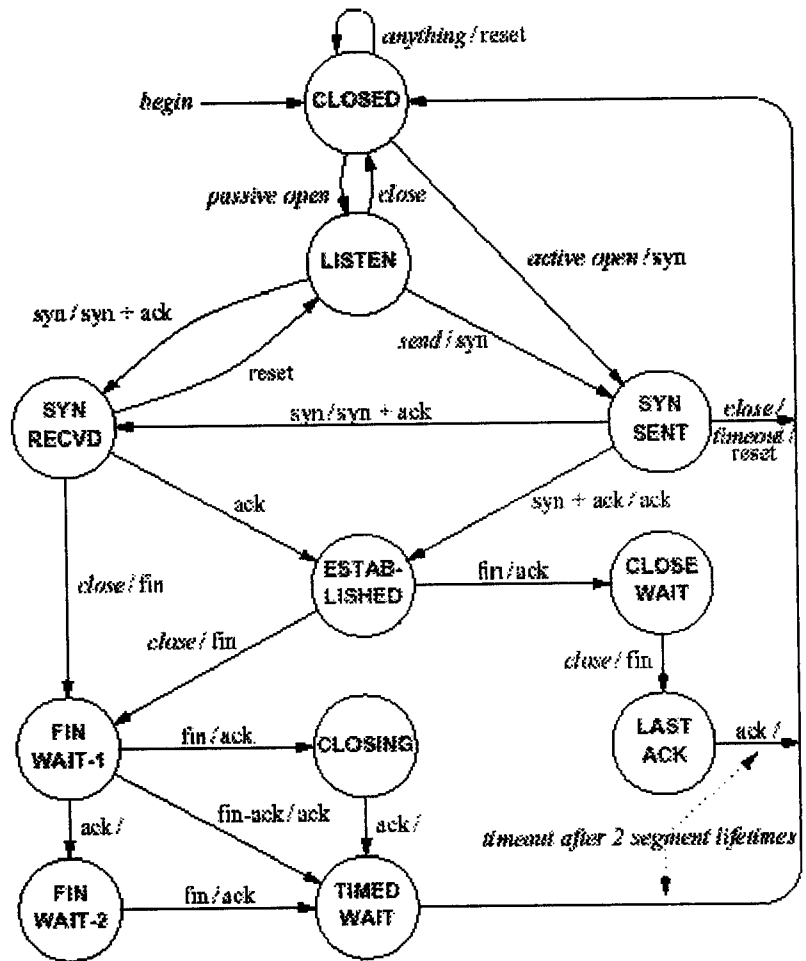


Figure 13.15 The TCP finite state machine. Each endpoint begins in the *closed* state. Labels on transitions show the input that caused the transition followed by the output if any.

Example State Transition:

To understand the TCP finite state machine, consider an example of the three-way handshake used to establish a connection between a client and a server. Both the client & server will establish an end point for communication, and both will have a copy of finite state machine. The server begins first by issuing a passive-open operation, which causes the server's finite state machine to enter into listen state. The server waits in the Listen state until a client contacts it. When the client issues an active open, it causes TCP software on it's machine to send a SYN segment to the server and to enter the SYN-SENT state.

When the server which is waiting in LISTEN state, receives a SYN segment, it replies with an SYN plus an ACK segment, creates a new TCB, and places the new TCB in SYN-RECEIVED state. When the SYN plus ACK segment arrives at the client, the client TCP replies with an ACK, and moves from the SYN-SENT state to the ESTABLISHED state. Finally when the Client's ACK arrives at the newly created TCB, it also moves to the ESTABLISHED state, which allows data transfer to proceed.

Because the TCP state machine contains few states, specifies few transitions among the states, provides complex operations, our implementation uses a procedure-oriented approach. Each state procedure handles incoming segments. It must accommodate request to abort(e.g.; RESET), special request to start (SYN), and request to shut down (FIN).

The requirement that TCP accepts segments out of order complicates most state procedures. For example, TCP may receive a request for shut down (FIN) before all data arrives. Or it may receive a segment carrying data before a segment that completes the 3 way handshake used to establish the connection. Our implementation accommodates out of delivery by recording startup and shutdown events in the TCB and checking them as each segment arrives.

TCP Output Processing:

TCP output is complex because it interacts closely with TCP input & timer events, all of which occur concurrently. For example, when the output process sends a segment, it must schedule a retransmission event. Later, if the retransmission timer expires, the Timer process must send the segment. Meanwhile the application program may generate data, causing TCP to send more segments, or acknowledgements may arrive, causing TCP to cancel retransmission events. However, the underlying IP protocol may drop, delay, or deliver segments out of order, events may not occur in the expected order. Even if data arrives at remote site, an acknowledgement may be lost. Because the remote site may receive data out of order, a single ACK may acknowledge receipt of many segments. Furthermore a site may receive the FIN for the connection before it has received all data segments, so retransmission may be necessary even after an application closes a connection. Thus the correct response to an input or output event depends on the history of previous events.

TCP output as a process:

Using a separate TCP output process helps separation execution of input, timer and output functions, and allows them to operate concurrently. For example, a retransmission timer may expire and trigger retransmission, while the input process is sending an acknowledgment in response to an incoming segment. The interaction may be especially complex because a TCP segment may carry acknowledgements along with data. If each procedure that needs output acts independently, TCP generates unnecessary traffic.

To co-ordinate output, our implementation uses a single process to handle output and makes all interaction message driven. When a procedure need to generate output, it places information in the TCB and sends a message to the TCP output process.

System Implementation

4. SYSTEM IMPLEMENTATION

The unit testing has been done separately for each of the modules. Both white box and black box testing techniques were employed. The testing was done with maximum number of test cases. The test cases were carefully designed during the process of coding and design. The test cases covered almost every part of the code to make the modules fault proof. The errors uncovered were rectified, and the modules were re-tested with all the test cases to ensure that the corrective measures taken did not cause any inadvertent effects. Some samples are shown in the table below.

4.1 Testing and Test Plan

The modules, their functionality and interaction between modules are tested in the integrated subsystems. Testing was also carried out to investigate the changes made in the individual modules, to check and find the weakness in them.

Various testing strategies like unit testing, integration testing, validation testing and system testing were applied. The testing techniques like white-box testing, basis path testing, control structure testing, and black box testing ensured successful testing of the system.

4.2 Testing Methods

Unit Testing

The unit testing has been done separately for each of the modules. Both white box and black box testing techniques were employed. The testing was done with maximum number of test cases. The test cases were carefully designed during the process of coding and design. The test cases covered almost every part of the code to make the modules fault proof. The errors uncovered were rectified, and the modules were re-tested with all the test cases to ensure that the corrective measures taken did not cause any inadvertent effects.

Integration Testing

The integration testing was done when various modules of the system were integrated together. Top-Down approach was followed during the process of integration. Further, *depth first* integration was carried out. The black box testing technique was employed throughout this phase. Stubs were created as required to ensure an incremental approach, and later, on completion of each set of tests, stubs were appropriately replaced with real modules. Finally, Regression testing was conducted to ensure that the errors rectified were successfully working, and they do not cause any adverse effects.

During testing, each interface provided by the system was supplied with variety of arguments respective to it. The results were as expected for all combinations of the arguments.

Validation Testing

The Validation testing was carried out at the culmination of the integration testing. Validation was done against the *validation criteria* specified in the software requirements specification. It was done as a series of black-box tests that demonstrated conformity with requirements. It ensured that all functional requirements are satisfied; all performance requirements are achieved; documentation is correct and human-engineered; and other requirements are met.

Performance & Stress Testing

As a part of system testing, performance tests were conducted along with stress testing. The stress and performance of the system was calculated based on the traffic on the network, system load and other system resources. It was found that the system was working fine with no stress and slowed down in small fractions of time when the traffic and CPU load was increased. The variation of other available system resources like memory, disk-space did not make any significant performance degradation in the system.

Recovery Testing

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. Many different test cases were designed and tested, for example killing the *dashboard* explicitly, forcing one of the workstation involved to crash, dismantling the network connections abruptly, flooding the network. In all the cases the system responded as expected, sending appropriate notifications and error messages.

5. LIMITATIONS AND FUTURE ENHANCEMENTS

The following enhancements can be done to our implementation of TCP:

- **TCP Timer Management:**
 - Determining the remaining time for an event
 - Inserting a Timer Event.
 - Deleting a Timer Event.

- **TCP Flow Control and Adaptive Retransmission:**
 - Congestion avoidance and control
 - Slow-Start and congestion avoidance

- **TCP Urgent Data Processing**
 - Sending Urgent Data

Conclusion

6. CONCLUSION

Developing source code for any protocol throws light on the minute details and intricacies of the protocol. It also helps us to understand the Xinu system environment and its system calls.

Understanding even simple ideas such as how TCP buffers data can help us design, implement & debug applications. Though not built for a commercial purpose, this project has helped us learn, understand and implement the protocols in TCP/IP.

References

7. REFERENCES

Books:

- Andrew S. Tanenbaum, "*Modern Operating Systems*", Prentice Hall of India, 1996
- Douglas E. Comer., "*Internetworking with TCP/IP*" Volume 1, Third Edition, Prentice Hall of India, 2001
- W. Richard Stevens., "*Unix Network Programming*" Volume 1, Second Edition, Addison Wesley, 1999

Web Sites:

- www.cs.purdue.edu - December 2003
- www.searchnetworking.com - January 2004
- www.whatis.techtarget.com - January - 2004

Appendix-A

Sample Code

SAMPLE CODE

```
arpfind(u_char *pra, u_short prtype, struct netif *pni)
{
    struct arprent *pae;
    int i;
    for (i=0; i<ARP_TSIZE; ++i) {
        pae = &arptable[i];
        if (pae->ae_state == AS_FREE)
            continue;
        if (pae->ae_prtype == prtype &&
            pae->ae_pni == pni &&
            BLKEQU(pae->ae_pra, pra, pae->ae_prlen))
            return pae;
    }
    return 0;
}
```

```
void arpsend(struct arprent *pae)
{
    struct ep *pep;
    struct netif *pni;
    if (pae->ae_queue == EMPTY)
        return;
    pni = pae->ae_pni;
    while (pep = (struct ep *)deq(pae->ae_queue))
        netwrite(pni, pep, pep->ep_len);
}
```



```
    freeq(pae->ae_queue);
    pae->ae_queue = EMPTY;
}
```

```
void arpinit()
```

```
{
    int    i;
    rarpsem = screate(1);
    rarppid = BADPID;
    for (i=0; i<ARP_TSIZE; ++i)
        arptable[i].ae_state = AS_FREE;
}
```

```
void arpsend(struct arpentry *pae)
```

```
{
    struct ep    *pep;
    struct netif *pni;
    if (pae->ae_queue == EMPTY)
        return;
    pni = pae->ae_pni;
    while (pep = (struct ep *)deq(pae->ae_queue))
        netwrite(pni, pep, pep->ep_len);
    freeq(pae->ae_queue);
    pae->ae_queue = EMPTY;
}
```

```
int ipputp(unsigned ifn, IPaddr nh, struct ep *pep)
```

```
{
    struct netif *pni = &nif[ifn];
```

```

struct ip      *pip;
int           hlen, maxdlen, tosend, offset, offindg;
if (pni->ni_state == NIS_DOWN) {
    freebuf(peg);
    return SYSERR;
}
pip = (struct ip *)peg->ep_data;
if (pip->ip_len <= pni->ni_mtu) {
    pep->ep_nexthop = nh;
    pip->ip_cksum = 0;
    iph2net(pip);
    pep->ep_order &= ~EPO_IP;
    pip->ip_cksum = cksum((WORD *)pip, IP_HLEN(pip));
    return netwrite(pni, pep, EP_HLEN+net2hs(pip->ip_len));
}
/* else, we need to fragment it */
if (pip->ip_fragoff & IP_DF) {
    IpFragFails++;
    icmp(ICT_DESTUR, ICC_FNADF, pip->ip_src, pep, 0);
    return OK;
}
maxdlen = (pni->ni_mtu - IP_HLEN(pip)) &~ 7;
offset = 0;
offindg = (pip->ip_fragoff & IP_FRAGOFF)<<3;
tosend = pip->ip_len - IP_HLEN(pip);
while (tosend > maxdlen) {
    if (ipfsend(pni,nh,peg,offset,maxdlen,offindg) != OK) {
        IpOutDiscards++;
        freebuf(peg);
        return SYSERR;
    }
}

```

```

    IpFragCreates++;
    tosend -= maxdlen;
    offset += maxdlen;
    offindg += maxdlen;
}

```

```

IpFragOKs++;
IpFragCreates++;

```

```

hlen = ipfhcopy(pep, pep, offindg);
pip = (struct ip *)pep->ep_data;
/* slide the residual down */
memcpy(&pep->ep_data[hlen], &pep->ep_data[IP_HLEN(pip)+offset],
       tosend);
/* keep MF, if this was a frag to start with */
pip->ip_fragoff = (pip->ip_fragoff & IP_MF)|(offindg>>3);
pip->ip_len = tosend + hlen;
pip->ip_cksum = 0;
iph2net(pip);
pep->ep_order &= ~EPO_IP;
pip->ip_cksum = cksum((WORD *)pip, hlen);
pep->ep_nexthop = nh;
return netwrite(pni, pep, EP_HLEN+net2hs(pip->ip_len));
}

```

```

int upalloc(void)

```

```

{
    struct upq  *pup;
    int         i;
    wait(udpmutex);
    for (i=0 ; i<UPPS ; i++) {

```

```

pup = &upqs[i];
if (!pup->up_valid) {
    pup->up_valid = TRUE;
    pup->up_port = 0;
    pup->up_pid = BADPID;
    pup->up_xport = pcreate(UPPLEN);
    signal(udpmutex);
    return i;
}
}
signal(udpmutex);
return SYSERR;
}

```

```

tcballloc(void)
{
    struct tcb    *ptcb;
    int          slot;
    wait(tcps_tmutex);
    /* look for a free TCB */
    for (ptcb=&tcbbtab[0], slot=0; slot<Ntcp; ++slot, ++ptcb)
        if (ptcb->tcb_state == TCPS_FREE)
            break;
    if (slot < Ntcp) {
        ptcb->tcb_state = TCPS_CLOSED;
        ptcb->tcb_mutex = screate(0);
    } else
        ptcb = (struct tcb *)SYSERR;
    signal(tcps_tmutex);
    return ptcb;
}

```

```
}
```

```
int tcpabort(struct tcb *ptcb, int error)
```

```
{
    tcpkilltimers(ptcb);
    ptcb->tcb_flags |= TCBF_RDONE|TCBF_SDONE;
    ptcb->tcb_error = error;
    tcpwakeup(READERS|WRITERS, ptcb);
    return OK;
}
```

```
int tcpclosing(struct tcb *ptcb, struct ep *pep)
```

```
{
    struct ip    *pip = (struct ip *)pep->ep_data;
    struct tcp   *ptcp = (struct tcp *)pip->ip_data;
    if (ptcp->tcp_code & TCPF_RST)
        return tcbdealloc(ptcb);
    if (ptcp->tcp_code & TCPF_SYN) {
        tcpreset(pep);
        return tcbdealloc(ptcb);
    }
    tcpacked(ptcb, pep);
    if ((ptcb->tcb_code & TCPF_FIN) == 0) {
        ptcb->tcb_state = TCPS_TIMEWAIT;
        signal(ptcb->tcb_ocsem);          /* wake closer */
        tcpwait(ptcb);
    }
    return OK;
}
```