UNIT 9.
Networks

Structure

9.0 Introduction
9.1 Objectives
9.2 An Overview of TCP/IP Networking
9.3 The Linux TCP/IP Networking Layers
9.4 The BSD Socket Interface
9.5 The INET Socket Layer
    9.5.1 Creating a BSD Socket
    9.5.2 Binding an Address to an INET BSD Socket
    9.5.3 Making a Connection on an INET BSD Socket
    9.5.4 Listening on an INET BSD Socket
    9.5.5 Accepting Connection Requests
9.6 The IP Layer
    9.6.1 Socket Buffers
    9.6.2 Receiving IP Packets
    9.6.3 Sending IP Packets
    9.6.4 Data Fragmentation
9.7 The Address Resolution Protocol (ARP)
9.8 Summary
9.9 Check Your Progress

9.0 Introduction

Networking and Linux are terms that are almost synonymous. In a very real sense Linux is a product of the Internet or World Wide Web (WWW). Its developers and users use the web to exchange information ideas, code, and Linux itself is often used to support the networking needs of organizations. This unit describes how Linux supports the network protocols known collectively as TCP/IP.

The TCP/IP protocols were designed to support communications between computers connected to the ARPANET, an American research network funded by the US government. The ARPANET pioneered networking concepts such as packet switching and protocol layering where one protocol uses the services of another. ARPANET was retired in 1988 but its successors (Internet) have grown even larger. What is now known as the World Wide Web grew from the ARPANET and is itself supported by the TCP/IP protocols. Unix ™ was extensively used on the ARPANET and the first released networking version of Unix ™ was 4.3 BSD. Linux's networking implementation is modeled on 4.3 BSD in that it supports BSD sockets (with some extensions) and the full range of TCP/IP networking. This programming interface was chosen because of its popularity and to help applications be portable between Linux and other Unix ™ platforms.

9.1 Objectives

At the end of this unit, You would be able to

- Undestand the main principles of TCP/IP networking
- Know different Networking layers
- Define Network supporting interfaces
- Explain layer that supports the internet address
- Describe sending & receiving of IP Packets
- Explain Address Resolution Protocol

9.2 An Overview of TCP/IP Networking

This section gives an overview of the main principles of TCP/IP networking. It is not meant to be an exhaustive description, for that I suggest that you read . In an IP network every machine is assigned an IP address, this is a 32 bit number that uniquely identifies the machine. The WWW is a very large, and growing, IP network and every machine that is connected to it has to have a unique IP address assigned to it. IP addresses are represented by four numbers separated by dots, for example, `16.42.0.9`. This IP address is actually in two parts, the *network* address and the *host* address. The sizes of these parts may vary (there are several classes of IP addresses) but using `16.42.0.9` as an example, the network address would be `16.42` and the host address `0.9`. The host address is further subdivided into a *subnetwork* and a *host* address. Again, using `16.42.0.9` as an example, the subnetwork address would be `16.42.0` and the host address 16.42.0.9. This subdivision of the IP address allows organizations to subdivide their networks. For example, `16.42` could be the network address of the ACME Computer Company; `16.42.0` would be subnet `0` and `16.42.1` would be subnet `1`. These subnets might be in separate buildings, perhaps connected by leased telephone lines or even microwave links. IP addresses are assigned by the network administrator and having IP subnetworks is a good way of distributing the administration of the network. IP subnet administrators are free to allocate IP addresses within their IP subnetworks.

Generally though, IP addresses are somewhat hard to remember. Names are much easier. `linux.acme.com` is much easier to remember than `16.42.0.9` but there must be some mechanism to convert the network names into an IP address. These names can be statically specified in the `/etc/hosts` file or Linux can ask a Distributed Name Server (DNS server) to resolve the name for it. In this case the local host must know the IP address of one or more DNS servers and these are specified in `/etc/resolv.conf`.

Whenever you connect to another machine, say when reading a web page, its IP address is used to exchange data with that machine. This data is contained in IP packets each of which have an IP header containing the IP addresses of the source and destination machine's IP addresses, a checksum and other useful information. The checksum is derived from the data in the IP packet and allows the receiver of IP packets to tell if the IP packet was corrupted during transmission, perhaps by a noisy telephone line. The data transmitted by an application may have been broken down into smaller packets which are easier to handle. The size of the IP data packets varies depending on the connection media; ethernet packets are generally bigger than PPP packets. The destination host must reassemble the data packets before giving the data to the receiving application. You can see this fragmentation and reassembly of data graphically if you access a web page containing a lot of graphical images via a moderately slow serial link.

Hosts connected to the same IP subnet can send IP packets directly to each other, all other IP packets will be sent to a special host, a gateway. Gateways (or routers) are connected to more than one IP subnet and they will resend IP packets received on one subnet, but destined for another onwards. For example, if subnets `16.42.1.0` and `16.42.0.0` are connected together by a gateway then any packets sent from subnet `0` to subnet `1` would have to be directed to the gateway so that it could route them. The local host builds up routing tables which allow it to route IP packets to the correct machine. For every IP destination there is an entry in the routing tables which tells Linux which host to send IP packets to in order that they reach their destination. These routing tables are dynamic and change over time as applications use the network and as the network topology changes.

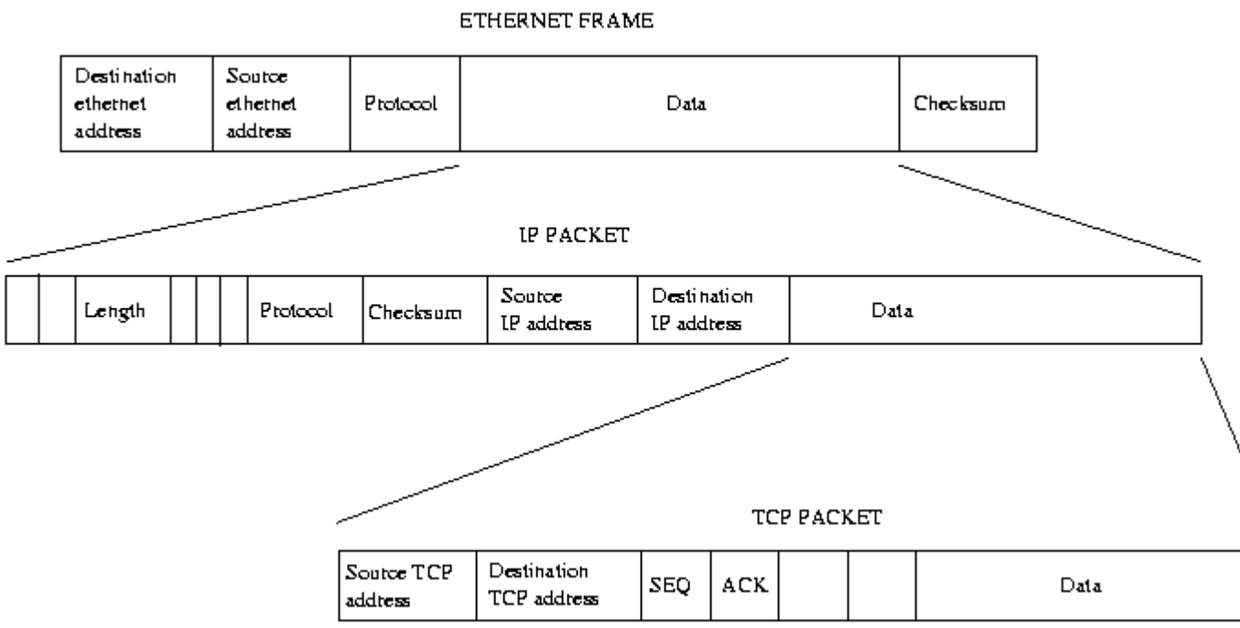

Figure 9.1: TCP/IP Protocol Layers

The IP protocol is a transport layer that is used by other protocols to carry their data. The Transmission Control Protocol (TCP) is a reliable end to end protocol that uses IP to transmit and receive its own packets. Just as IP packets have their own header, TCP has its own header. TCP is a connection based protocol where two networking applications are connected by a single, virtual connection even though there may be many subnetworks, gateways and routers between them. TCP reliably transmits and receives data between the two applications and guarantees that there will be no lost or duplicated data. When TCP transmits its packet using IP, the data contained within the IP packet is the TCP packet itself. The IP layer on each communicating host is responsible for transmitting and receiving IP packets. User Datagram Protocol (UDP) also uses the IP layer to transport its packets, unlike TCP, UDP is not a reliable protocol but offers a datagram service. This use of IP by other protocols means that when IP packets are received the receiving IP layer must know which upper protocol layer to give the data contained in this IP packet to. To facilitate this every IP packet header has a byte containing a protocol identifier. When TCP asks the IP layer to transmit an IP packet , that IP packet's header states that it contains a TCP packet. The receiving IP layer uses that protocol identifier to decide which layer to pass the received data up to, in this case the TCP layer. When applications communicate via TCP/IP they must specify not only

the target's IP address but also the *port* address of the application. A port address uniquely identifies an application and standard network applications use standard port addresses; for example, web servers use port 80. These registered port addresses can be seen in `/etc/services`.

This layering of protocols does not stop with TCP, UDP and IP. The IP protocol layer itself uses many different physical media to transport IP packets to other IP hosts. These media may themselves add their own protocol headers. One such example is the ethernet layer, but PPP and SLIP are others. An ethernet network allows many hosts to be simultaneously connected to a single physical cable. Every transmitted ethernet frame can be seen by all connected hosts and so every ethernet device has a unique address. Any ethernet frame transmitted to that address will be received by the addressed host but ignored by all the other hosts connected to the network. These unique addresses are built into each ethernet device when they are manufactured and it is usually kept in an SROM on the ethernet card. Ethernet addresses are 6 bytes long, an example would be `08-00-2b-00-49-A4`. Some ethernet addresses are reserved for multicast purposes and ethernet frames sent with these destination addresses will be received by all hosts on the network. As ethernet frames can carry many different protocols (as data) they, like IP packets, contain a protocol identifier in their headers. This allows the ethernet layer to correctly receive IP packets and to pass them onto the IP layer.

In order to send an IP packet via a multi-connection protocol such as ethernet, the IP layer must find the ethernet address of the IP host. This is because IP addresses are simply an addressing concept, the ethernet devices themselves have their own physical addresses. IP addresses on the other hand can be assigned and reassigned by network administrators at will but the network hardware responds only to ethernet frames with its own physical address or to special multicast addresses which all machines must receive. Linux uses the Address Resolution Protocol (or ARP) to allow machines to translate IP addresses into real hardware addresses such as ethernet addresses. A host wishing to know the hardware address associated with an IP address sends an ARP request packet containing the IP address that it wishes translating to all nodes on the network by sending it to a multicast address. The target host that owns the IP address, responds with an ARP reply that contains its physical hardware address. ARP is not just restricted to ethernet devices, it can resolve IP addresses for other physical media, for example FDDI. Those network devices that cannot ARP are marked so that Linux does not attempt to ARP. There is also the reverse function, Reverse ARP or RARP, which translates phsyical network addresses into IP addresses. This is used by gateways, which respond to ARP requests on behalf of IP addresses that are in the remote network.

9.3 The Linux TCP/IP Networking Layers

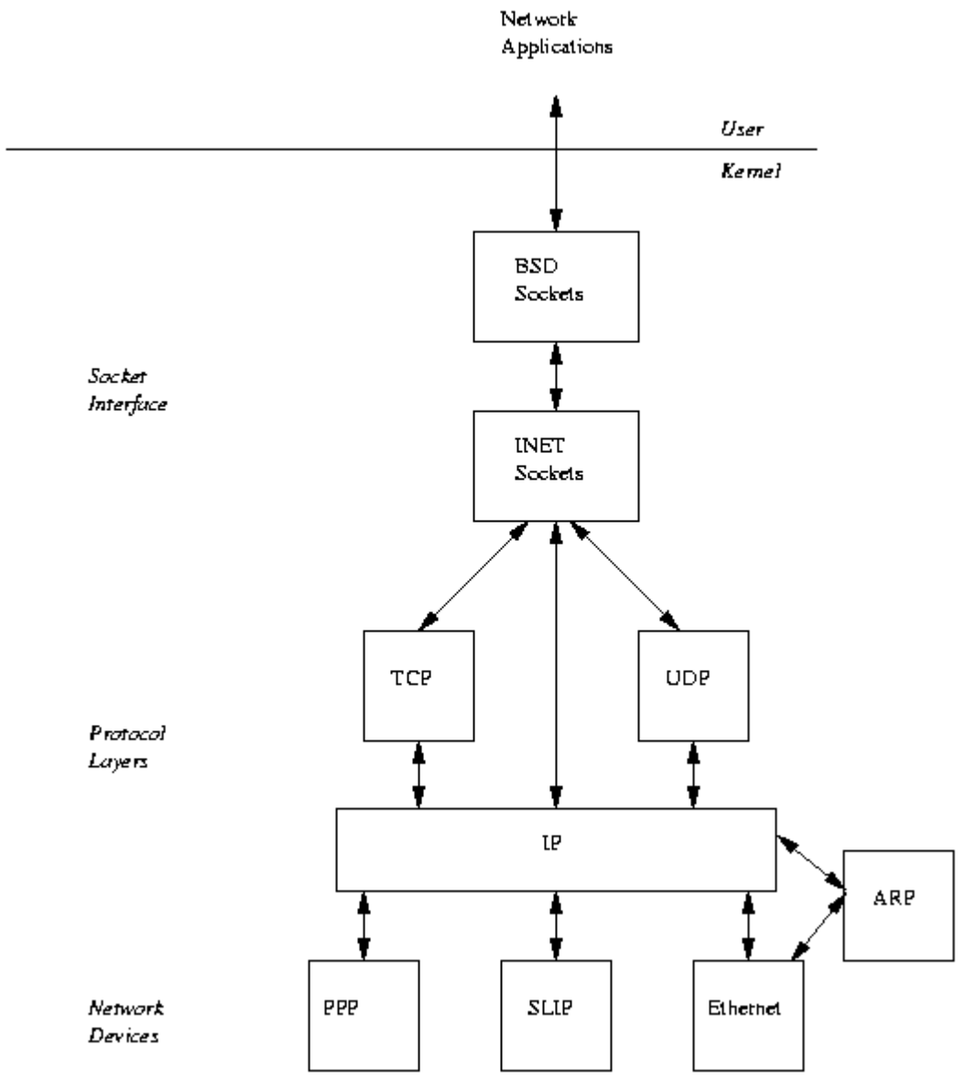

Figure 9.2: Linux Networking Layers

Just like the network protocols themselves, Figure 9.2 shows that Linux implements the internet protocol address family as a series of connected layers of software. BSD sockets are supported by a generic socket management software concerned only with BSD sockets. Supporting this is the INET socket layer, this manages the communication end points for the IP based protocols TCP and UDP. UDP (User Datagram Protocol) is a connectionless protocol whereas TCP (Transmission Control Protocol) is a reliable end to end protocol. When UDP packets are transmitted, Linux neither knows nor cares if they arrive safely at their destination. TCP packets are numbered and both ends of the TCP connection make sure that transmitted data is received correctly. The IP layer contains code implementing the Internet Protocol. This code prepends IP headers to transmitted data and understands how to route incoming IP packets to either the TCP or UDP layers. Underneath the IP layer, supporting all of Linux's networking are the network devices, for example PPP and ethernet. Network devices do not always represent physical devices; some like the loopback device are

purely software devices. Unlike standard Linux devices that are created via the mknod command, network devices appear only if the underlying software has found and initialized them. You will only see `/dev/eth0` when you have built a kernel with the appropriate ethernet device driver in it. The ARP protocol sits between the IP layer and the protocols that support ARPing for addresses.

9.4 The BSD Socket Interface

This is a general interface which not only supports various forms of networking but is also an inter-process communications mechanism. A socket describes one end of a communications link, two communicating processes would each have a socket describing their end of the communication link between them. Sockets could be thought of as a special case of pipes but, unlike pipes, sockets have no limit on the amount of data that they can contain. Linux supports several classes of socket and these are known as *address families*. This is because each class has its own method of addressing its communications. Linux supports the following socket address families or domains:

UNIX	Unix domain sockets,
INET	The Internet address family supports communications via TCP/IP protocols
AX25	Amateur radio X25
IPX	Novell IPX
APPLETALK	Appletalk DDP
X25	X25

There are several socket types and these represent the type of service that supports the connection. Not all address families support all types of service. Linux BSD sockets support a number of socket types:

Stream
: These sockets provide reliable two way sequenced data streams with a guarantee that data cannot be lost, corrupted or duplicated in transit. Stream sockets are supported by the TCP protocol of the Internet (INET) address family.

Datagram
: These sockets also provide two way data transfer but, unlike stream sockets, there is no guarantee that the messages will arrive. Even if they do arrive there is no guarantee that they will arrive in order or even not be duplicated or corrupted. This type of socket is supported by the UDP protocol of the Internet address family.

Raw
: This allows processes direct (hence ``raw'') access to the underlying protocols. It is, for example, possible to open a raw socket to an ethernet device and see raw IP data traffic.

Reliable Delivered Messages
: These are very like datagram sockets but the data is guaranteed to arrive.

Sequenced Packets
: These are like stream sockets except that the data packet sizes are fixed.

Packet
: This is not a standard BSD socket type, it is a Linux specific extension that allows processes to access packets directly at the device level.

Processes that communicate using sockets use a client server model. A server provides a service and clients make use of that service. One example would be a Web Server, which provides web pages and a web client, or browser, which reads those pages.

A server using sockets, first creates a socket and then binds a name to it. The format of this name is dependent on the socket's address family and it is, in effect, the local address of the server. The socket's name or address is specified using the `sockaddr` data structure. An INET socket would have an IP port address bound to it. The registered port numbers can be seen in `/etc/services`; for example, the port number for a web server is 80. Having bound an address to the socket, the server then listens for incoming connection requests specifying the bound address. The originator of the request, the client, creates a socket and makes a connection request on it, specifying the target address of the server. For an INET socket the address of the server is its IP address and its port number. These incoming requests must find their way up through the various protocol layers and then wait on the server's listening socket. Once the server has received the incoming request it either accepts or rejects it. If the incoming request is to be accepted, the server must create a new socket to accept it on. Once a socket has been used for listening for incoming connection requests it cannot be used to support a connection. With the connection established both ends are free to send and receive data. Finally, when the connection is no longer needed it can be shutdown. Care is taken to ensure that data packets in transit are correctly dealt with.

The exact meaning of operations on a BSD socket depends on its underlying address family. Setting up TCP/IP connections is very different from setting up an amateur radio X.25 connection. Like the virtual filesystem, Linux abstracts the socket interface with the BSD socket layer being concerned with the BSD socket interface to the application programs which is in turn supported by independent address family specific software. At kernel initialization time, the address families built into the kernel register themselves with the BSD socket interface. Later on, as applications create and use BSD sockets, an association is made between the BSD socket and its supporting address family. This association is made via cross-linking data structures and tables of address family specific support routines. For example there is an address family specific socket creation routine which the BSD socket interface uses when an application creates a new socket.

When the kernel is configured, a number of address families and protocols are built into the `protocols` vector. Each is represented by its name, for example ``INET'' and the address of its initialization routine. When the socket interface is initialized at boot time each protocol's initialization routine is called. For the socket address families this results in them registering a set of protocol operations. This is a set of routines, each of which performs a a particular operation specific to that address family. The registered protocol operations are kept in the `pops` vector, a vector of pointers to `proto_ops` data structures.

The `proto_ops` data structure consists of the address family type and a set of pointers to socket operation routines specific to a particular address family. The `pops` vector is indexed by the address family identifier, for example the Internet address family identifier (AF_INET is 2).

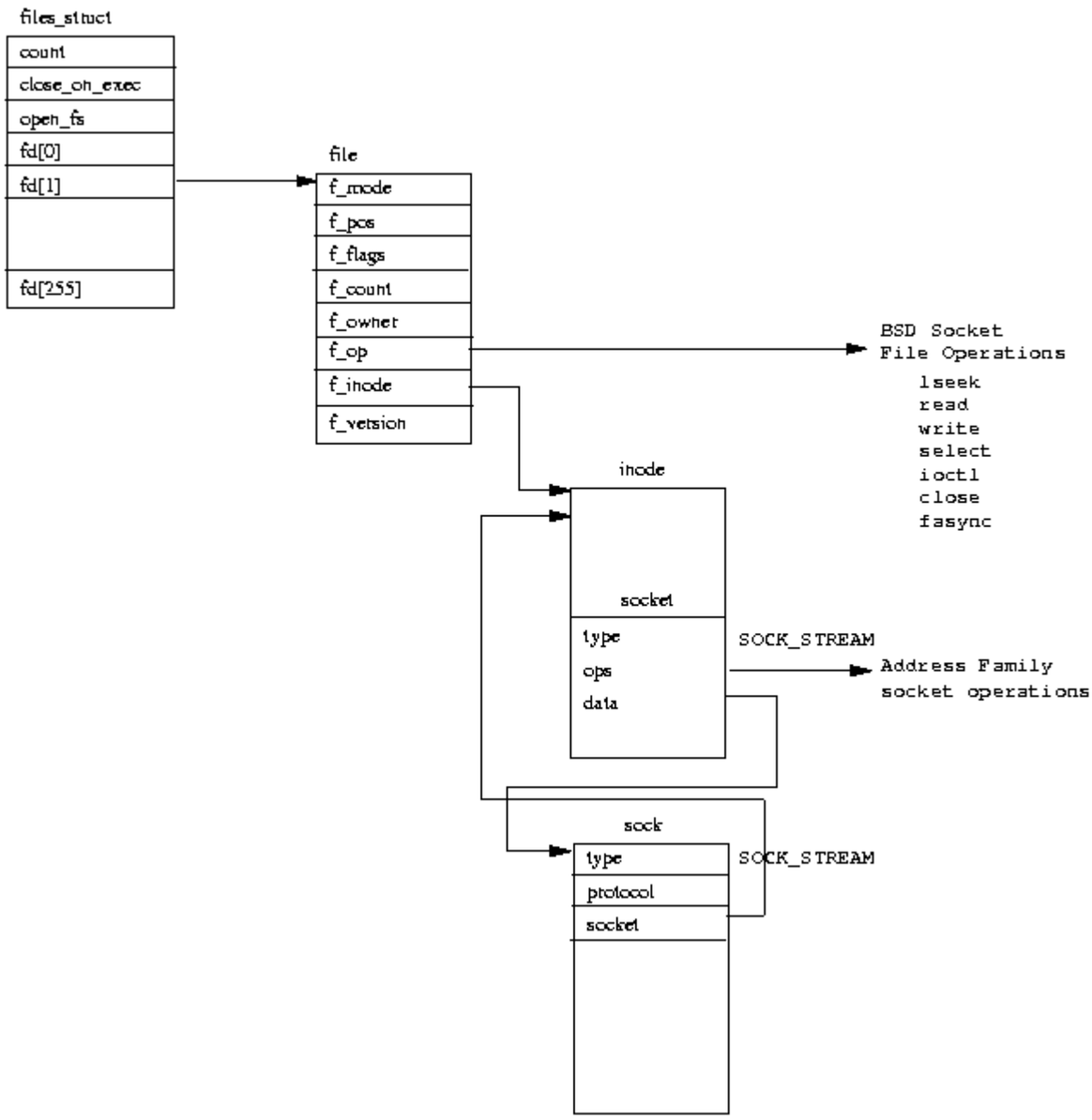

Figure 9.3: Linux BSD Socket Data Structures

9.5 The INET Socket Layer

The INET socket layer supports the internet address family which contains the TCP/IP protocols. As discussed above, these protocols are layered, one protocol using the services of another. Linux's TCP/IP code and data structures reflect this layering. Its interface with the BSD socket layer is through the set of Internet address family socket operations which it registers with the BSD socket layer during network initialization. These are kept in the `pops` vector along with the other registered address families. The BSD socket layer calls the INET layer socket support routines from the registered INET `proto_ops` data structure to perform work for it. For example a BSD socket create request that gives the address family as INET will use the underlying INET socket create

function. The BSD socket layer passes the `socket` data structure representing the BSD socket to the INET layer in each of these operations. Rather than clutter the BSD `socket` wiht TCP/IP specific information, the INET socket layer uses its own data structure, the `sock` which it links to the BSD `socket` data structure. This linkage can be seen in Figure 9.3. It links the `sock` data structure to the BSD `socket` data structure using the `data` pointer in the BSD `socket`. This means that subsequent INET socket calls can easily retrieve the `sock` data structure. The `sock` data structure's protocol operations pointer is also set up at creation time and it depends on the protocol requested. If TCP is requested, then the `sock` data structure's protocol operations pointer will point to the set of TCP protocol operations needed for a TCP connection.

9.5.1 Creating a BSD Socket

The system call to create a new socket passes identifiers for its address family, socket type and protocol.

Firstly the requested address family is used to search the `pops` vector for a matching address family. It may be that a particular address family is implemented as a kernel module and, in this case, the `kerneld` daemon must load the module before we can continue. A new `socket` data structure is allocated to represent the BSD socket. Actually the `socket` data structure is physically part of the VFS `inode` data structure and allocating a socket really means allocating a VFS `inode`. This may seem strange unless you consider that sockets can be operated on in just the same way that ordinairy files can. As all files are represented by a VFS `inode` data structure, then in order to support file operations, BSD sockets must also be represented by a VFS `inode` data structure.

The newly created BSD `socket` data structure contains a pointer to the address family specific socket routines and this is set to the `proto_ops` data structure retrieved from the `pops` vector. Its type is set to the sccket type requested; one of SOCK_STREAM, SOCK_DGRAM and so on. The address family specific creation routine is called using the address kept in the `proto_ops` data structure.

A free file descriptor is allocated from the current processes `fd` vector and the `file` data structure that it points at is initialized. This includes setting the file operations pointer to point to the set of BSD socket file operations supported by the BSD socket interface. Any future operations will be directed to the socket interface and it will in turn pass them to the supporting address family by calling its address family operation routines.

9.5.2 Binding an Address to an INET BSD Socket

In order to be able to listen for incoming internet connection requests, each server must create an INET BSD socket and bind its address to it. The bind operation is mostly handled within the INET socket layer with some support from the underlying TCP and UDP protocol layers. The socket having an address bound to cannot be being used for any other communication. This means that the `socket`'s state must be `TCP_CLOSE`. The `sockaddr` pass to the bind operation contains the IP address to be bound to and, optionally, a port number. Normally the IP address bound to would be one that has been assigned to a network device that supports the INET address family and whose interface is up and able to be used. You can see which network interfaces are currently active in the system by using the ifconfig command. The IP address may also be the IP broadcast address of either all 1's or all 0's. These are special addresses that mean ``send to

everybody". The IP address could also be specified as any IP address if the machine is acting as a transparent proxy or firewall, but only processes with superuser privileges can bind to any IP address. The IP address bound to is saved in the `sock` data structure in the `recv_addr` and `saddr` fields. These are used in hash lookups and as the sending IP address respectively. The port number is optional and if it is not specified the supporting network is asked for a free one. By convention, port numbers less than 1024 cannot be used by processes without superuser privileges. If the underlying network does allocate a port number it always allocates ones greater than 1024.

As packets are being received by the underlying network devices they must be routed to the correct INET and BSD sockets so that they can be processed. For this reason UDP and TCP maintain hash tables which are used to lookup the addresses within incoming IP messages and direct them to the correct `socket`/`sock` pair. TCP is a connection oriented protocol and so there is more information involved in processing TCP packets than there is in processing UDP packets.

UDP maintains a hash table of allocated UDP ports, the `udp_hash` table. This consists of pointers to `sock` data structures indexed by a hash function based on the port number. As the UDP hash table is much smaller than the number of permissible port numbers (`udp_hash` is only 128 or `UDP_HTABLE_SIZE` entries long) some entries in the table point to a chain of `sock` data structures linked together using each `sock`'s `next` pointer.

TCP is much more complex as it maintains several hash tables. However, TCP does not actually add the binding `sock` data stucture into its hash tables during the bind operation, it merely checks that the port number requested is not currently being used. The `sock` data structure is added to TCP's hash tables during the *listen* operation.

9.5.3 Making a Connection on an INET BSD Socket

Once a socket has been created and, provided it has not been used to listen for inbound connection requests, it can be used to make outbound connection requests. For connectionless protocols like UDP this socket operation does not do a whole lot but for connection orientated protocols like TCP it involves building a virtual circuit between two applications.

An outbound connection can only be made on an INET BSD socket that is in the right state; that is to say one that does not already have a connection established and one that is not being used for listening for inbound connections. This means that the BSD `socket` data structure must be in state `SS_UNCONNECTED`. The UDP protocol does not establish virtual connections between applications, any messages sent are datagrams, one off messages that may or may not reach their destinations. It does, however, support the *connect* BSD socket operation. A connection operation on a UDP INET BSD socket simply sets up the addresses of the remote application; its IP address and its IP port number. Additionally it sets up a cache of the routing table entry so that UDP packets sent on this BSD socket do not need to check the routing database again (unless this route becomes invalid). The cached routing information is pointed at from the `ip_route_cache` pointer in the INET `sock` data structure. If no addressing information is given, this cached routing and IP addressing information will be automatically be used for messages sent using this BSD socket. UDP moves the `sock`'s state to `TCP_ESTABLISHED`.

For a connect operation on a TCP BSD socket, TCP must build a TCP message containing the connection information and send it to IP destination given. The TCP message contains information about the connection, a unique starting message sequence number, the maximum sized message that can be managed by the initiating host, the transmit and receive window size and so on. Within TCP all messages are numbered and the initial sequence number is used as the first message number. Linux chooses a reasonably random value to avoid malicious protocol attacks. Every message transmitted by one end of the TCP connection and successfully received by the other is acknowledged to say that it arrived successfully and uncorrupted. Unacknowledges messages will be retransmitted. The transmit and receive window size is the number of outstanding messages that there can be without an acknowledgement being sent. The maximum message size is based on the network device that is being used at the initiating end of the request. If the receiving end's network device supports smaller maximum message sizes then the connection will use the minimum of the two. The application making the outbound TCP connection request must now wait for a response from the target application to accept or reject the connection request. As the TCP `sock` is now expecting incoming messages, it is added to the `tcp_listening_hash` so that incoming TCP messages can be directed to this `sock` data structure. TCP also starts timers so that the outbound connection request can be timed out if the target application does not respond to the request.

9.5.4 Listening on an INET BSD Socket

Once a socket has had an address bound to it, it may listen for incoming connection requests specifying the bound addresses. A network application can listen on a socket without first binding an address to it; in this case the INET socket layer finds an unused port number (for this protocol) and automatically binds it to the socket. The listen socket function moves the socket into state `TCP_LISTEN` and does any network specific work needed to allow incoming connections.

For UDP sockets, changing the socket's state is enough but TCP now adds the socket's `sock` data structure into two hash tables as it is now active. These are the `tcp_bound_hash` table and the `tcp_listening_hash`. Both are indexed via a hash function based on the IP port number.

Whenever an incoming TCP connection request is received for an active listening socket, TCP builds a new `sock` data structure to represent it. This `sock` data structure will become the bottom half of the TCP connection when it is eventually accepted. It also clones the incoming `sk_buff` containing the connection request and queues it onto the `receive_queue` for the listening `sock` data structure. The clone `sk_buff` contains a pointer to the newly created `sock` data structure.

9.5.5 Accepting Connection Requests

UDP does not support the concept of connections, accepting INET socket connection requests only applies to the TCP protocol as an accept operation on a listening socket causes a new `socket` data structure to be cloned from the original listening `socket`. The accept operation is then passed to the supporting protocol layer, in this case INET to accept any incoming connection requests. The INET protocol layer will fail the accept operation if the underlying protocol, say UDP, does not support connections. Otherwise the accept operation is passed through to the real protocol, in this case TCP. The accept operation can be either blocking or non-blocking. In the non-blocking case if there are no incoming connections to accept, the accept operation will fail and the newly

created `socket` data structure will be thrown away. In the blocking case the network application performing the accept operation will be added to a wait queue and then suspended until a TCP connection request is received. Once a connection request has been received the `sk_buff` containing the request is discarded and the `sock` data structure is returned to the INET socket layer where it is linked to the new `socket` data structure created earlier. The file descriptor (`fd`) number of the new `socket` is returned to the network application, and the application can then use that file descriptor in socket operations on the newly created INET BSD socket.

9.6 The IP Layer

9.6.1 Socket Buffers

One of the problems of having many layers of network protocols, each one using the services of another, is that each protocol needs to add protocol headers and tails to data as it is transmitted and to remove them as it processes received data. This make passing data buffers between the protocols difficult as each layer needs to find where its particular protocol headers and tails are. One solution is to copy buffers at each layer but that would be inefficient. Instead, Linux uses socket buffers or `sk_buffs` to pass data between the protocol layers and the network device drivers. `sk_buffs` contain pointer and length fields that allow each protocol layer to manipulate the application data via standard functions or “methods".

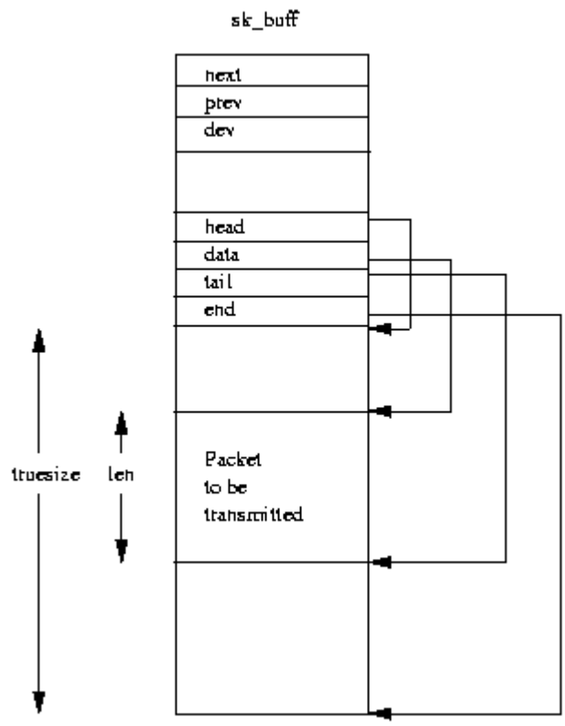

Figure 9.4: The Socket Buffer (sk_buff)

Figure 9.4 shows the `sk_buff` data structure; each `sk_buff` has a block of data associated with it. The `sk_buff` has four data pointers, which are used to manipulate and manage the socket buffer's data:

head
: points to the start of the data area in memory. This is fixed when the `sk_buff` and its associated data block is allocated,

data
: points at the current start of the protocol data. This pointer varies depending on the protocol layer that currently owns the `sk_buff`,

tail
: points at the current end of the protocol data. Again, this pointer varies depending on the owning protocol layer,

end
: points at the end of the data area in memory. This is fixed when the `sk_buff` is allocated.

There are two length fields `len` and `truesize`, which describe the length of the current protocol packet and the total size of the data buffer respectively. The `sk_buff` handling code provides standard mechanisms for adding and removing protocol headers and tails to the application data. These safely manipulate the `data`, `tail` and `len` fields in the `sk_buff`:

push
: This moves the `data` pointer towards the start of the data area and increments the `len` field. This is used when adding data or protocol headers to the start of the data to be transmitted,

pull
: This moves the `data` pointer away from the start, towards the end of the data area and decrements the `len` field. This is used when removing data or protocol headers from the start of the data that has been received,

put
: This moves the `tail` pointer towards the end of the data area and increments the `len` field. This is used when adding data or protocol information to the end of the data to be transmitted,

trim
: This moves the `tail` pointer towards the start of the data area and decrements the `len` field. This is used when removing data or protocol tails from the received packet.

The `sk_buff` data structure also contains pointers that are used as it is stored in doubly linked circular lists of `sk_buff`'s during processing. There are generic `sk_buff` routines for adding `sk_buffs` to the front and back of these lists and for removing them.

9.6.2 Receiving IP Packets

Linux's network drivers built are into the kernel and initialized. This results in a series of `device` data structures linked together in the `dev_base` list. Each `device` data structure describes its device and provides a set of callback routines that the network protocol layers call when they need the network driver to perform work. These functions are mostly concerned with transmitting data and with the network device's addresses. When a network device receives packets from its network it must convert the received data into `sk_buff` data structures. These received `sk_buff`'s are added onto the `backlog` queue by the network drivers as they are received.

If the `backlog` queue grows too large, then the received `sk_buff`'s are discarded. The network bottom half is flagged as ready to run as there is work to do.

When the network bottom half handler is run by the scheduler it processes any network packets waiting to be transmitted before processing the `backlog` queue of `sk_buff`'s determining which protocol layer to pass the received packets to.

As the Linux networking layers were initialized, each protocol registered itself by adding a `packet_type` data structure onto either the `ptype_all` list or into the `ptype_base` hash table. The `packet_type` data structure contains the protocol type, a pointer to a network device, a pointer to the protocol's receive data processing routine and, finally, a pointer to the next `packet_type` data structure in the list or hash chain. The `ptype_all` chain is used to snoop all packets being received from any network device and is not normally used. The `ptype_base` hash table is hashed by protocol identifier and is used to decide which protocol should receive the incoming network packet. The network bottom half matches the protocol types of incoming `sk_buff`'s against one or more of the `packet_type` entries in either table. The protocol may match more than one entry, for example when snooping all network traffic, and in this case the `sk_buff` will be cloned. The `sk_buff` is passed to the matching protocol's handling routine.

9.6.3 Sending IP Packets

Packets are transmitted by applications exchanging data or else they are generated by the network protocols as they support established connections or connections being established. Whichever way the data is generated, an `sk_buff` is built to contain the data and various headers are added by the protocol layers as it passes through them.

The `sk_buff` needs to be passed to a network device to be transmitted. First though the protocol, for example IP, needs to decide which network device to use. This depends on the best route for the packet. For computers connected by modem to a single network, say via the PPP protocol, the routing choice is easy. The packet should either be sent to the local host via the loopback device or to the gateway at the end of the PPP modem connection. For computers connected to an ethernet the choices are harder as there are many computers connected to the network.

For every IP packet transmitted, IP uses the routing tables to resolve the route for the destination IP address. Each IP destination successfully looked up in the routing tables returns a `rtable` data structure describing the route to use. This includes the source IP address to use, the address of the network `device` data structure and, sometimes, a prebuilt hardware header. This hardware header is network device specific and contains the source and destination physical addresses and other media specific information. If the network device is an ethernet device, the hardware header would be as shown in Figure 9.1 and the source and destination addresses would be physical ethernet addresses. The hardware header is cached with the route because it must be appended to each IP packet transmitted on this route and constructing it takes time. The hardware header may contain physical addresses that have to be resolved using the ARP protocol. In this case the outgoing packet is stalled until the address has been resolved. Once it has been resolved and the hardware header built, the hardware header is cached so that future IP packets sent using this interface do not have to ARP.

9.6.4 Data Fragmentation

Every network device has a maximum packet size and it cannot transmit or receive a data packet bigger than this. The IP protocol allows for this and will fragment data into smaller units to fit into the packet size that the network device can handle. The

IP protocol header includes a fragment field which contains a flag and the fragment offset.

When an IP packet is ready to be transmited, IP finds the network device to send the IP packet out on. This device is found from the IP routing tables. Each `device` has a field describing its maximum transfer unit (in bytes), this is the `mtu` field. If the device's mtu is smaller than the packet size of the IP packet that is waiting to be transmitted, then the IP packet must be broken down into smaller (mtu sized) fragments. Each fragment is represented by an `sk_buff`; its IP header marked to show that it is a fragment and what offset into the data this IP packet contains. The last packet is marked as being the last IP fragment. If, during the fragmentation, IP cannot allocate an `sk_buff`, the transmit will fail.

Receiving IP fragments is a little more difficult than sending them because the IP fragments can be received in any order and they must all be received before they can be reassembled. Each time an IP packet is received it is checked to see if it is an IP fragment. The first time that the fragment of a message is received, IP creates a new `ipq` data structure, and this is linked into the `ipqueue` list of IP fragments awaiting recombination. As more IP fragments are received, the correct `ipq` data structure is found and a new `ipfrag` data structure is created to describe this fragment. Each `ipq` data structure uniquely describes a fragmented IP receive frame with its source and destination IP addresses, the upper layer protocol identifier and the identifier for this IP frame. When all of the fragments have been received, they are combined into a single `sk_buff` and passed up to the next protocol level to be processed. Each `ipq` contains a timer that is restarted each time a valid fragment is received. If this timer expires, the `ipq` data structure and its `ipfrag`'s are dismantled and the message is presumed to have been lost in transit. It is then up to the higher level protocols to retransmit the message.

9.7 The Address Resolution Protocol (ARP)

The Address Resolution Protocol's role is to provide translations of IP addresses into physical hardware addresses such as ethernet addresses. IP needs this translation just before it passes the data (in the form of an `sk_buff`) to the device driver for transmission.

It performs various checks to see if this device needs a hardware header and, if it does, if the hardware header for the packet needs to be rebuilt. Linux caches hardware headers to avoid frequent rebuilding of them. If the hardware header needs rebuilding, it calls the device specific hardware header rebuilding routine. All ethernet devices use the same generic header rebuilding routine, which in turn uses the ARP services to translate the destination IP address into a physical address.

The ARP protocol itself is very simple and consists of two message types, an ARP request and an ARP reply. The ARP request contains the IP address that needs translating and the reply (hopefully) contains the translated IP address, the hardware address. The ARP request is broadcast to all hosts connected to the network, so, for an ethernet network, all of the machines connected to the ethernet will see the ARP request. The machine that owns the IP address in the request will respond to the ARP request with an ARP reply containing its own physical address.

The ARP protocol layer in Linux is built around a table of `arp_table` data structures which each describe an IP to physical address translation. These entries are created as IP addresses need to be translated and removed as they become stale over time. Each `arp_table` data structure has the following fields:

last used	the time that this ARP entry was last used,
last updated	the time that this ARP entry was last updated,
flags	these describe this entry's state, if it is complete and so on,
IP address	The IP address that this entry describes
hardware address	The translated hardware address
hardware header	This is a pointer to a cached hardware header,
timer	This is a `timer_list` entry used to time out ARP requests that do not get a response,
Retries	The number of times that this ARP request has been retried,
`sk_buff` queue	List of `sk_buff` entries waiting for this IP address to be resolved

The ARP table consists of a table of pointers (the `arp_tables` vector) to chains of `arp_table` entries. The entries are cached to speed up access to them, each entry is found by taking the last two bytes of its IP address to generate an index into the table and then following the chain of entries until the correct one is found. Linux also caches prebuilt hardware headers off the `arp_table` entries in the form of `hh_cache` data structures.

When an IP address translation is requested and there is no corresponding `arp_table` entry, ARP must send an ARP request message. It creates a new `arp_table` entry in the table and queues the `sk_buff` containing the network packet that needs the address translation on the `sk_buff` queue of the new entry. It sends out an ARP request and sets the ARP expiry timer running. If there is no response then ARP will retry the request a number of times and if there is still no response ARP will remove the `arp_table` entry. Any `sk_buff` data structures queued waiting for the IP address to be translated will be notified and it is up to the protocol layer that is transmitting them to cope with this failure. UDP does not care about lost packets but TCP will attempt to retransmit on an established TCP link. If the owner of the IP address responds with its hardware address, the `arp_table` entry is marked as complete and any queued `sk_buff`'s will be removed from the queue and will go on to be transmitted. The hardware address is written into the hardware header of each `sk_buff`.

The ARP protocol layer must also respond to ARP requests that specfy its IP address. It registers its protocol type (`ETH_P_ARP`), generating a `packet_type` data structure. This means that it will be passed all ARP packets that are received by the network devices. As well as ARP replies, this includes ARP requests. It generates an ARP reply using the hardware address kept in the receiving device's `device` data structure.

Network topologies can change over time and IP addresses can be reassigned to different hardware addresses. For example, some dial up services assign an IP address as each connection is established. In order that the ARP table contains up to date entries, ARP runs a periodic timer which looks through all of the `arp_table` entries to see which have timed out. It is very careful not to remove entries that contain one or more cached hardware headers. Removing these entries is dangerous as other data structures rely on them. Some `arp_table` entries are permanent and these are marked so that they will not be deallocated. The ARP table cannot be allowed to grow too large; each `arp_table` entry consumes some kernel memory. Whenever the a new entry needs to be allocated and the ARP table has reached its maximum size the table is pruned by searching out the oldest entries and removing them.

9.8 Summary

Linux offers a complete implementation of Transmission Control Protocol/Internet Protocol (TCP/IP), the protocol used extensively on the Internet and that is commonly found in local area networks involving UNIX machines. All you need to create a network, or to add your existing machine to a TCP/IP network, is a network card or network interface and some modifications to files already on your Linux system.

TCP/IP is an open networking protocol, which simply means that the technical description of all aspects of the protocol have been published. They are available for anyone to implement on their hardware and software. This open nature has helped make TCP/IP very popular. Versions of TCP/IP are now available for practically every hardware and software platform in existence, which has helped make TCP/IP the most widely used networking protocol in the world. The advantage of TCP/IP for a network operating system is simple: Interconnectivity is possible for any type of operating system and hardware platform that you might want to add.

TCP/IP is not a single protocol but a set of more than a dozen protocols. Each protocol within the TCP/IP family is dedicated to a different task. All the protocols that make up TCP/IP use the primary components of TCP/IP to send packets of data.

Transmission Control Protocol and Internet Protocol are two of the primary protocols in the TCP/IP family. The different protocols and services that make up the TCP/IP family can be grouped according to their purposes. The groups and their protocols are the following:

Transport: These protocols control the movement of data between two machines.

> TCP (Transmission Control Protocol): A connection-based service, meaning that the sending and receiving machines communicate with each other through a stream of messages. TCP has message delivery assurance routines incorporated into it.
>
> UDP (User Datagram Protocol): A connectionless service, meaning that the data is sent without the sending and receiving machines being in contact with each other. It's like sending snail-mail (regular postal service) with an address but no guarantee it will arrive.

Routing: These protocols handle the addressing of the data and determine the best routing to the destination. They also handle the way large messages are broken up and reassembled at the destination.

> IP (Internet Protocol): Handles the actual transmission of data.
>
> ICMP (Internet Control Message Protocol): Handles status messages for IP, such as errors and network changes that can affect routing.
>
> RIP (Routing Information Protocol): One of several protocols that determine the best routing method.
>
> OSPF (Open Shortest Path First): An alternative protocol for determining routing.

Network Addresses: These services handle the way machines are addressed, both by a unique number and a more common symbolic name.

ARP (Address Resolution Protocol): Determines the unique network hardware addresses of machines on the network based on their IP address.

DNS (Domain Name System): Determines numeric addresses from machine names.

RARP (Reverse Address Resolution Protocol): Determines IP addresses of machines on the network based on their network address (the opposite of ARP).

BOOTP (Boot Protocol): This starts up a network machine by reading the boot information from a server. BOOTP is commonly used for diskless workstations.

User Services: These are applications users have access to.

FTP (File Transfer Protocol): This protocol efficiently transfers files from one machine to another. FTP uses TCP as the transport.

TFTP (Trivial File Transfer Protocol): A simple file transfer method that uses UDP as the transport.

TELNET: Allows remote logins so that a user on one machine can connect to another machine and behave as though they are sitting at the remote machine's keyboard.

Gateway Protocols: These services help the network communicate routing and status information, as well as handle data for local networks.

EGP (Exterior Gateway Protocol): Transfers routing information for external networks.

GGP (Gateway-to-Gateway Protocol): Transfers routing information between Internet gateways.

IGP (Interior Gateway Protocol): Transfers routing information for internal networks.

Others: These are services that don't fall into the categories just mentioned but that provide important services over a network.

NFS (Network File System): Allows directories on one machine to be mounted on another, and then accessed by users as though the directories were on the local machine.

NIS (Network Information Service): Maintains user accounts across networks, simplifying logins and password maintenance.

RPC (Remote Procedure Call): Allows remote applications to communicate with each other using function calls.

SMTP (Simple Mail Transfer Protocol): A protocol for transferring electronic mail between machines.

SNMP (Simple Network Management Protocol): An administrator's service that sends status messages about the network and devices attached to it.

All the TCP/IP protocol definitions are maintained by a standards body that is part of the Internet organization. Although changes to the protocols occasionally occur when new features or better methods of performing older functions are developed, the new versions are almost always backward compatible.

9.9 Check Your Progress

I. Choose apropriate answer

1. IP address can be obtained using _____.
   a. network address
   b host address
   c both a & b
   d none of the above.
2. Hosts connected to the different IP subnet can send IP packets through _____.
   a. Routers
   b. Gateways
   c. either a or b
   d. none of the above
3. _____ is a reliable end to end protocol that uses IP to transmit and receive its own packets.
   a. TCP
   b. Telnet
   c. FTP
   d. UDP
4. _____ allows machines to translate IP addresses into ethernet addresses
   a UDP
   b. ARP
   c. TCP
   d. IP
5. UDP is a _____ protocol.
   a. connectionless
   b. end to end protocol.
   c. connection-oriented
   d. both b & c

II. Say True or False

1. The TCP/IP protocols were designed to support communications between computers connected to the ARPANET or INTERNET. True/False

2. IP addresses are represented by six numbers separated by dots, for example, `16.42.0.99.1.3` True/False

3. Gateways (or routers) are connected to more than one IP subnet and they will resend IP packets received on one subnet, but destined for another onwards. True/False

4. TCP is a connection based protocol where two networking applications are connected by a single, virtual connection even though there may be many subnetworks, gateways and routers between them. True/False

5. Ethernet addresses are 4 bytes long, an example would be `08-00-2b-00`. True/False

6. A socket describes one end of a communications link, two communicating processes would each have a socket describing their end of the communication link between them. True/False

7. The socket having an address bound to can be being used for any other communication. True/False

8. TCP is a connection oriented protocol and so there is more information involved in processing TCP packets than there is in processing UDP packets. True/False

9. For every IP packet transmitted, IP uses the routing tables to resolve the route for the destination IP address. True/False

10. The Address Resolution Protocol's role is to provide translations of IP addresses into physical hardware addresses such as ethernet addresses. True/False

11. RARP, which translates phsyical network addresses into IP addresses. True/False

III. Essay type Questions

1. Illustrate IP address with an example also explain TCP/IP Protocol Layers in detail.
2. Explain Linux Networking Layer in brief.
3. Write a short note on BSD socket interface with an neat data structure.
4. How to Create,Bind , Connect ,Listen an INET BSD Socket.
5. Write a note on IP Layer with relevant data structure.
6. Describe how to send & receive IP packets.
7. Write a note on Address Resolution Protocol.

IV. Further readings and other activities

1. Get more information using man or info or any help command for
   hostname, domainname, ipconfig, ping, netconf,
2. Install a network comprising of two systems, one being the Linux OS and the other as Windows OS. Test the interconnection using the ping, ifconfig, winipcfg and other network related commands.
3. Get more information on networks through the text book,
   Title: Computer Networks
   Author: Tanenbaum.

Reference e-mails: raomvp@yahoo.com roopasindhe@lycos.com
URL / Web Site: http://www.raomvp.bravepages.com

Good-Luck