Limited-broadcast Address

The only address in the block **255.255.255.255/32** is called the *limited-broadcast* address. It is used whenever a router or a host needs to send a datagram to all devices in a network. The routers in the network, however, block the packet having this address as the destination; the packet cannot travel outside the network.

Loopback Address

The block **127.0.0.0/8** is called the *loopback* address. A packet with one of the addresses in this block as the destination address never leaves the host; it will remain in the host. Any address in the block is used to test a piece of software in the machine. For example, we can write a client and a server program in which one of the addresses in the block is used as the server address. We can test the programs using the same host to see if they work before running them on different computers.

Private Addresses

Four blocks are assigned as private addresses: 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, and 169.254.0.0/16. We will see the applications of these addresses when we discuss NAT later in the chapter.

Multicast Addresses

The block 224.0.0.0/4 is reserved for multicast addresses. We discuss these addresses later in the chapter.

18.4.4 Dynamic Host Configuration Protocol (DHCP)

We have seen that a large organization or an ISP can receive a block of addresses directly from ICANN and a small organization can receive a block of addresses from an ISP. After a block of addresses are assigned to an organization, the network administration can manually assign addresses to the individual hosts or routers. However, address assignment in an organization can be done automatically using the **Dynamic Host Configuration Protocol (DHCP).** DHCP is an application-layer program, using the client-server paradigm, that actually helps TCP/IP at the network layer.

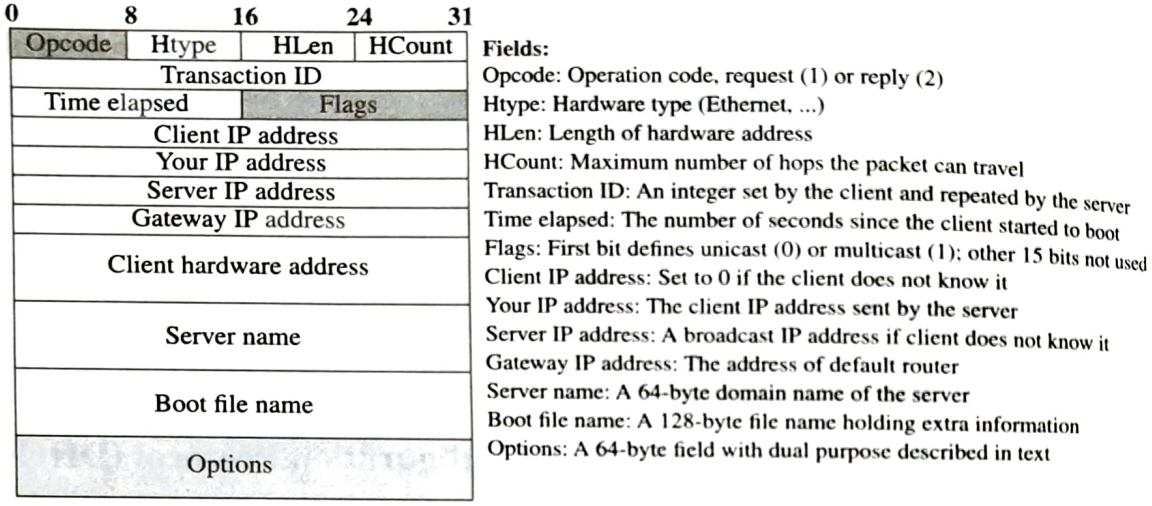
DHCP has found such widespread use in the Internet that it is often called a *plug-and-play protocol*. In can be used in many situations. A network manager can configure DHCP to assign permanent IP addresses to the host and routers. DHCP can also be configured to provide temporary, on demand, IP addresses to hosts. The second capability can provide a temporary IP address to a traveller to connect her laptop to the Internet while she is staying in the hotel. It also allows an ISP with 1000 granted addresses to provide services to 4000 households, assuming not more than one-forth of customers use the Internet at the same time.

In addition to its IP address, a computer also needs to know the network prefix (or address mask). Most computers also need two other pieces of information, such as the address of a default router to be able to communicate with other networks and the address of a name server to be able to use names instead of addresses, as we will see in Chapter 26. In other words, four pieces of information are normally needed: the computer address, the prefix, the address of a router, and the IP address of a name server. DHCP can be used to provide these pieces of information to the host.

DHCP Message Format

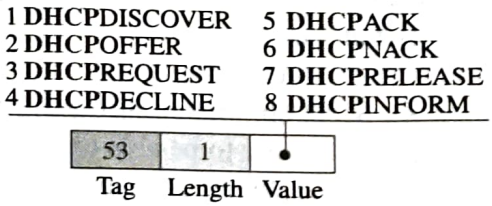
DHCP is a client-server protocol in which the client sends a request message and the server returns a response message. Before we discuss the operation of DHCP, let us show the general format of the DHCP message in Figure 18.25. Most of the fields are explained in the figure, but we need to discuss the option field, which plays a very important role in DHCP.

Figure 18.25 *DHCP message format*



The 64-byte option field has a dual purpose. It can carry either additional information or some specific vendor information. The server uses a number, called a **magic cookie,** in the format of an IP address with the value of 99.130.83.99. When the client finishes reading the message, it looks for this magic cookie. If present, the next 60 bytes are options. An option is composed of three fields: a 1-byte tag field, a 1-byte length field, and a variable-length value field. There are several tag fields that are mostly used by vendors. If the tag field is 53, the value field defines one of the 8 message types shown in Figure 18.26. We show how these message types are used by DHCP.

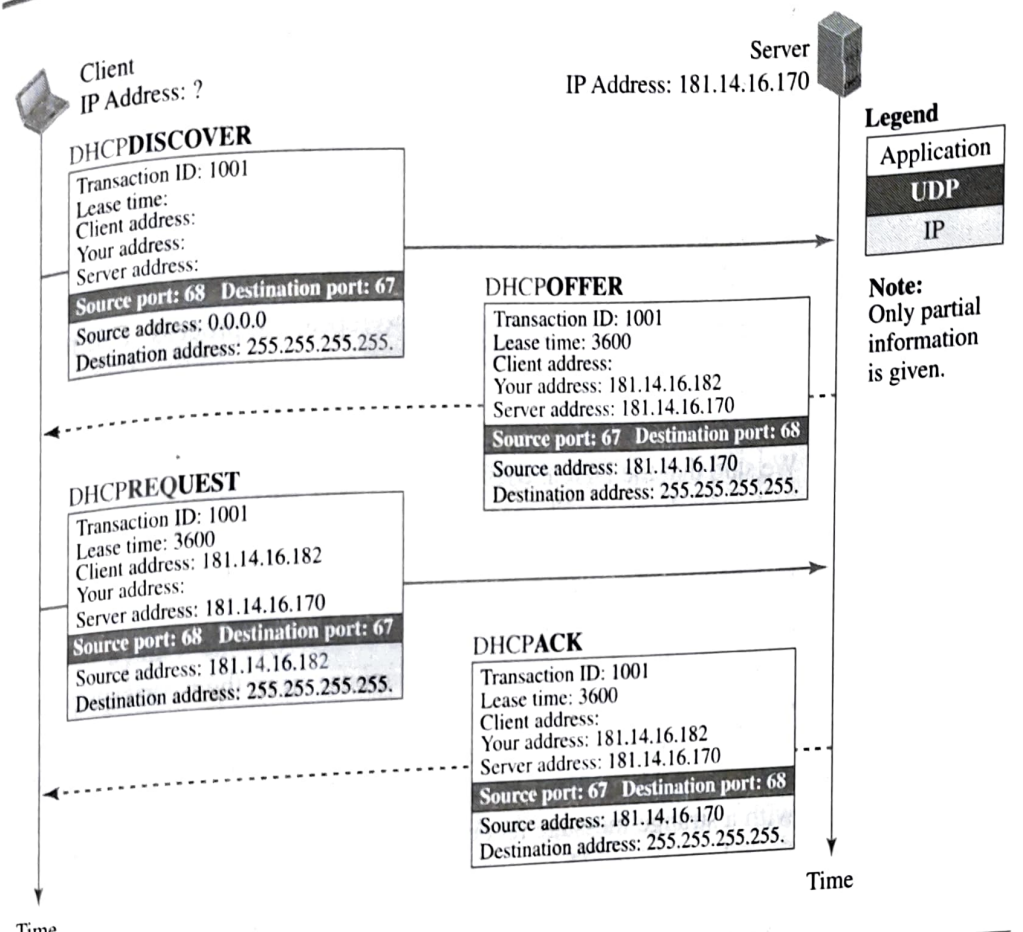
Figure 18.26 *Option format*



DHCP Operation

Figure 18.27 shows a simple scenario.

Figure 18.27 *Operation of DHCP*



1. The joining host creates a **DHCPDISCOVER** message in which only the transaction-ID field is set to a random number. No other field can be set because the host has no knowledge with which to do so. This message is encapsulated in a UDP user datagram with the source port set to 68 and the destination port set to 67. We will discuss the reason for using two well-known port numbers later. The user datagram is encapsulated in an IP datagram with the source address set to **0.0.0.0** ("this host") and the destination address set to **255.255.255.255** (broadcast address). The reason is that the joining host knows neither its own address nor the server address.
2. The DHCP server or servers (if more than one) responds with a **DHCPOFFER** message in which the your address field defines the offered IP address for the joining host and the server address field includes the IP address of the server. The message also includes the lease time for which the host can keep the IP address. This message is encapsulated in a user datagram with the same port numbers, but in the reverse order. The user datagram in turn is encapsulated in a datagram with the server address as the source IP address, but the destination address is a broadcast address, in which the server allows other DHCP servers to receive the offer and give a better offer if they can.

3. The joining host receives one or more offers and selects the best of them. The joining host then sends a **DHCPREQUEST** message to the server that has given the best offer. The fields with known value are set. The message is encapsulated in a user datagram with port numbers as the first message. The user datagram is encapsulated in an IP datagram with the source address set to the new client address, but the destination address still is set to the broadcast address to let the other servers know that their offer was not accepted.
4. Finally, the selected server responds with a **DHCPACK** message to the client if the offered IP address is valid. If the server cannot keep its offer (for example, if the address is offered to another host in between), the server sends a **DHCPNACK** message and the client needs to repeat the process. This message is also broadcast to let other servers know that the request is accepted or rejected.

Two Well-Known Ports

We said that the DHCP uses two well-known ports (68 and 67) instead of one well-known and one ephemeral. The reason for choosing the well-known port 68 instead of an ephemeral port for the client is that the response from the server to the client is broadcast. Remember that an IP datagram with the limited broadcast message is delivered to every host on the network. Now assume that a DHCP client and a DAYTIME client, for example, are both waiting to receive a response from their corresponding server and both have accidentally used the same temporary port number (56017, for example). Both hosts receive the response message from the DHCP server and deliver the message to their clients. The DHCP client processes the message; the DAYTIME client is totally confused with a strange message received. Using a well-known port number prevents this problem from happening. The response message from the DHCP server is not delivered to the DAYTIME client, which is running on the port number 56017, not 68. The temporary port numbers are selected from a different range than the well-known port numbers.

The curious reader may ask what happens if two DHCP clients are running at the same time. This can happen after a power failure and power restoration. In this case the messages can be distinguished by the value of the transaction ID, which separates each response from the other.

Using FTP

The server does not send all of the information that a client may need for joining the network. In the **DHCPACK** message, the server defines the pathname of a file in which the client can find complete information such as the address of the DNS server. The client can then use a file transfer protocol to obtain the rest of the needed information.

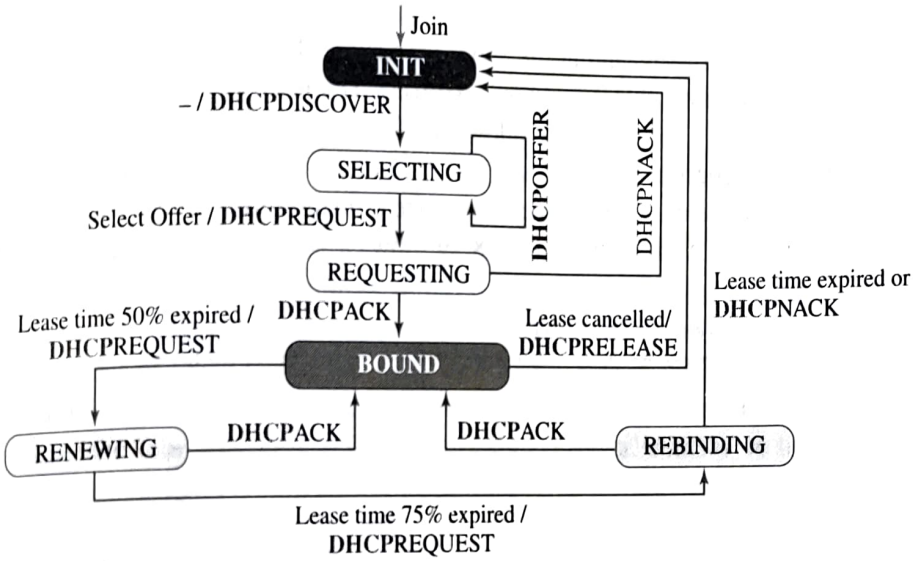
Error Control

DHCP uses the service of UDP, which is not reliable. To provide error control, DHCP uses two strategies. First, DHCP requires that UDP use the checksum. As we will see in Chapter 24, the use of the checksum in UDP is optional. Second, the DHCP client uses timers and a retransmission policy if it does not receive the DHCP reply to a request. However, to prevent a traffic jam when several hosts need to retransmit a request (for example, after a power failure), DHCP forces the client to use a random number to set its timers.

Transition States

The previous scenarios we discussed for the operation of the DHCP were very simple. To provide dynamic address allocation, the DHCP client acts as a state machine that performs transitions from one state to another depending on the messages it receives or sends. Figure 18.28 shows the transition diagram with the main states.

Figure 18.28 *FSM for the DHCP client*



When the DHCP client first starts, it is in the INIT state (initializing state). The client broadcasts a discover message. When it receives an offer, the client goes to the SELECTING state. While it is there, it may receive more offers. After it selects an offer, it sends a request message and goes to the REQUESTING state. If an ACK arrives while the client is in this state, it goes to the BOUND state and uses the IP address. When the lease is 50 percent expired, the client tries to renew it by moving to the RENEWING state. If the server renews the lease, the client moves to the BOUND state again. If the lease is not renewed and the lease time is 75 percent expired, the client moves to the REBINDING state. If the server agrees with the lease (ACK message arrives), the client moves to the BOUND state and continues using the IP address; otherwise, the client moves to the INIT state and requests another IP address. Note that the client can use the IP address only when it is in the BOUND, RENEWING, or REBINDING state. The above procedure requires that the client uses three timers: *renewal timer* (set to 50 percent of the lease time), *rebinding timer* (set to 75 percent of the lease time), and *expiration timer* (set to the lease time).

18.4.5 Network Address Resolution (NAT)

The distribution of addresses through ISPs has created a new problem. Assume that an ISP has granted a small range of addresses to a small business or a household. If the business grows or the household needs a larger range, the ISP may not be able to grant the demand because the addresses before and after the range may have already been allocated to other networks. In most situations, however, only a portion of computers in

The length field defines the length of the packet but does not include the padding. One to eight bytes of padding is added to the packet to make the attack on the security provision more difficult. The *cyclic redundancy check* (CRC) field is used for error detection. The type field designates the type of the packet used in different SSH protocols. The data field is the data transferred by the packet in different protocols.

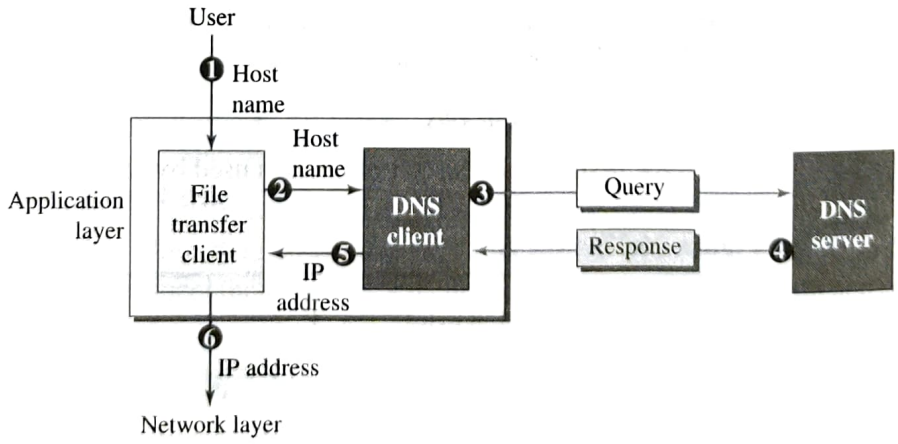
26.6 DOMAIN NAME SYSTEM (DNS)

The last client-server application program we discuss has been designed to help other application programs. To identify an entity, TCP/IP protocols use the IP address, which uniquely identifies the connection of a host to the Internet. However, people prefer to use names instead of numeric addresses. Therefore, the Internet needs to have a directory system that can map a name to an address. This is analogous to the telephone network. A telephone network is designed to use telephone numbers, not names. People can either keep a private file to map a name to the corresponding telephone number or can call the telephone directory to do so. We discuss how this directory system in the Internet can map names to IP addresses.

Since the Internet is so huge today, a central directory system cannot hold all the mapping. In addition, if the central computer fails, the whole communication network will collapse. A better solution is to distribute the information among many computers in the world. In this method, the host that needs mapping can contact the closest computer holding the needed information. This method is used by the **Domain Name System (DNS)**. We first discuss the concepts and ideas behind the DNS. We then describe the DNS protocol itself.

Figure 26.28 shows how TCP/IP uses a DNS client and a DNS server to map a name to an address. A user wants to use a file transfer client to access the corresponding file transfer server running on a remote host. The user knows only the file transfer

Figure 26.28 *Purpose of DNS*



server name, such as *afilesource.com*. However, the TCP/IP suite needs the IP address of the file transfer server to make the connection. The following six steps map the host name to an IP address:

1. The user passes the host name to the file transfer client.
2. The file transfer client passes the host name to the DNS client.
3. Each computer, after being booted, knows the address of one DNS server. The DNS client sends a message to a DNS server with a query that gives the file transfer server name using the known IP address of the DNS server.
4. The DNS server responds with the IP address of the desired file transfer server.
5. The DNS server passes the IP address to the file transfer client.
6. The file transfer client now uses the received IP address to access the file transfer server.

Note that the purpose of accessing the Internet is to make a connection between the file transfer client and server, but before this can happen, another connection needs to be made between the DNS client and DNS server. In other words, we need at least two connections in this case. The first is for mapping the name to an IP address; the second is for transferring files. We will see later that the mapping may need more than one connection.

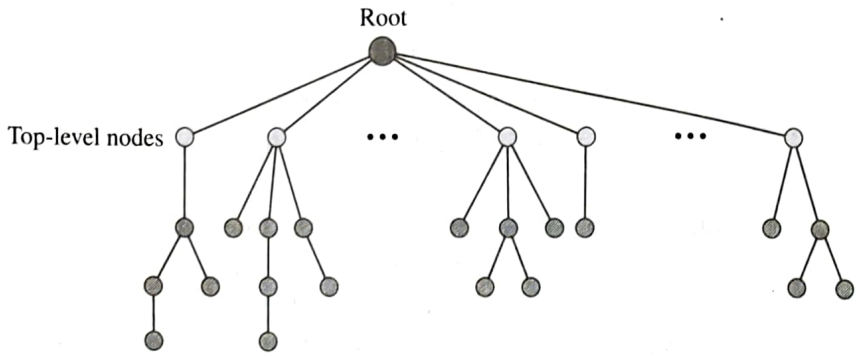
26.6.1 Name Space

To be unambiguous, the names assigned to machines must be carefully selected from a name space with complete control over the binding between the names and IP addresses. In other words, the names must be unique because the addresses are unique. A **name space** that maps each address to a unique name can be organized in two ways: flat or hierarchical. In a *flat name space,* a name is assigned to an address. A name in this space is a sequence of characters without structure. The names may or may not have a common section; if they do, it has no meaning. The main disadvantage of a flat name space is that it cannot be used in a large system such as the Internet because it must be centrally controlled to avoid ambiguity and duplication. In a *hierarchical name space,* each name is made of several parts. The first part can define the nature of the organization, the second part can define the name of an organization, the third part can define departments in the organization, and so on. In this case, the authority to assign and control the name spaces can be decentralized. A central authority can assign the part of the name that defines the nature of the organization and the name of the organization. The responsibility for the rest of the name can be given to the organization itself. The organization can add suffixes (or prefixes) to the name to define its host or resources. The management of the organization need not worry that the prefix chosen for a host is taken by another organization because, even if part of an address is the same, the whole address is different. For example, assume two organizations call one of their computers *caesar.* The first organization is given a name by the central authority, such as *first.com,* the second organization is given the name *second.com.* When each of these organizations adds the name *caesar* to the name they have already been given, the end result is two distinguishable names: *ceasar.first.com* and *ceasar.second.com.* The names are unique.

Domain Name Space

To have a hierarchical name space, a **domain name space** was designed. In this design the names are defined in an inverted-tree structure with the root at the top. The tree can have only 128 levels: level 0 (root) to level 127 (see Figure 26.29).

Figure 26.29 *Domain name space*



Label

Each node in the tree has a **label**, which is a string with a maximum of 63 characters. The root label is a null string (empty string). DNS requires that children of a node (nodes that branch from the same node) have different labels, which guarantees the uniqueness of the domain names.

Domain Name

Each node in the tree has a domain name. A full **domain name** is a sequence of labels separated by dots (.). The domain names are always read from the node up to the root. The last label is the label of the root (null). This means that a full domain name always ends in a null label, which means the last character is a dot because the null string is nothing. Figure 26.30 shows some domain names.

If a label is terminated by a null string, it is called a **fully qualified domain name (FQDN)**. The name must end with a null label, but because null means nothing, the label ends with a dot. If a label is not terminated by a null string, it is called a **partially qualified domain name (PQDN).** A PQDN starts from a node, but it does not reach the root. It is used when the name to be resolved belongs to the same site as the client. Here the resolver can supply the missing part, called the *suffix,* to create an FQDN.

Domain

A **domain** is a subtree of the domain name space. The name of the domain is the name of the node at the top of the subtree. Figure 26.31 shows some domains. Note that a domain may itself be divided into domains.

Distribution of Name Space

The information contained in the domain name space must be stored. However, it is very inefficient and also not reliable to have just one computer store such a huge

Figure 26.30 *Domain names and labels*

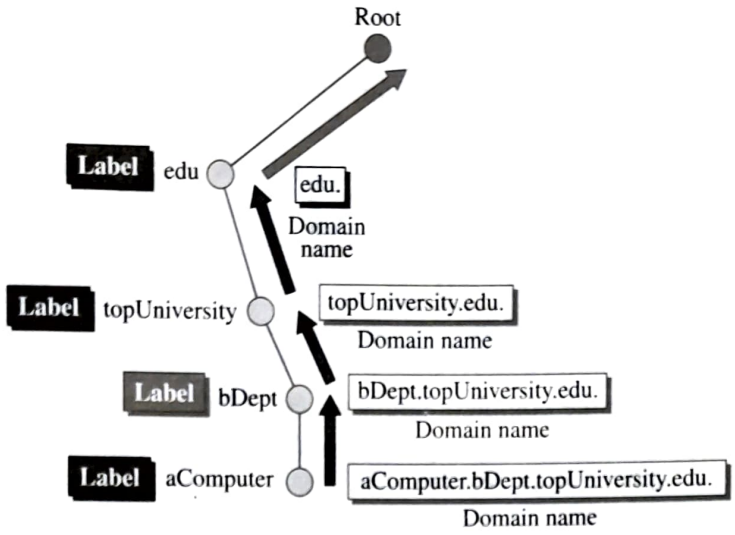
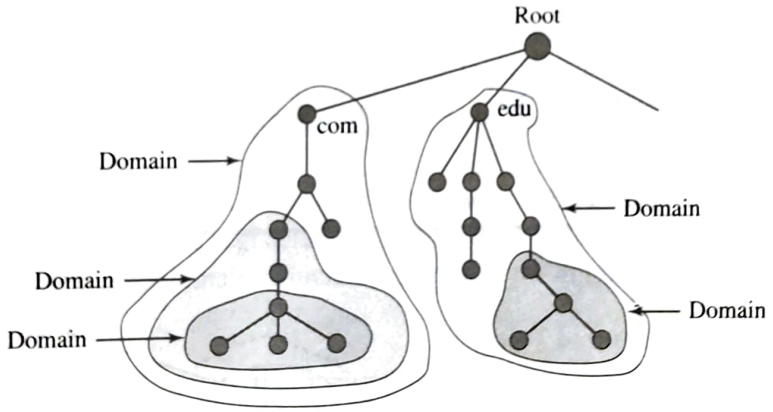


Figure 26.31 *Domains*

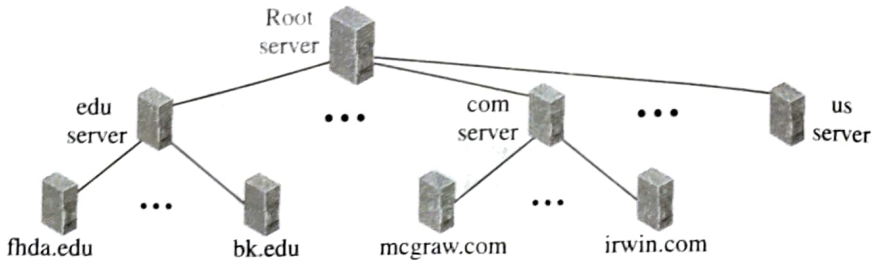


amount of information. It is inefficient because responding to requests from all over the world places a heavy load on the system. It is not reliable because any failure makes the data inaccessible.

Hierarchy of Name Servers

The solution to these problems is to distribute the information among many computers called ***DNS servers.*** One way to do this is to divide the whole space into many domains based on the first level. In other words, we let the root stand alone and create as many domains (subtrees) as there are first-level nodes. Because a domain created this way could be very large, DNS allows domains to be divided further into smaller domains (subdomains). Each server can be responsible (authoritative) for either a large or small domain. In other words, we have a hierarchy of servers in the same way that we have a hierarchy of names (see Figure 26.32).

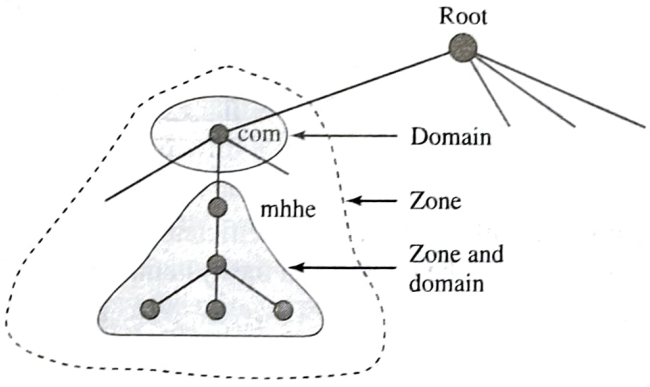
Figure 26.32 *Hierarchy of name servers*



Zone

Since the complete domain name hierarchy cannot be stored on a single server, it is divided among many servers. What a server is responsible for or has authority over is called a ***zone***. We can define a zone as a contiguous part of the entire tree. If a server accepts responsibility for a domain and does not divide the domain into smaller domains, the "domain" and the "zone" refer to the same thing. The server makes a database called a *zone file* and keeps all the information for every node under that domain. However, if a server divides its domain into subdomains and delegates part of its authority to other servers, "domain" and "zone" refer to different things. The information about the nodes in the subdomains is stored in the servers at the lower levels, with the original server keeping some sort of reference to these lower-level servers. Of course, the original server does not free itself from responsibility totally. It still has a zone, but the detailed information is kept by the lower-level servers (see Figure 26.33).

Figure 26.33 *Zone*



Root Server

A **root server** is a server whose zone consists of the whole tree. A root server usually does not store any information about domains but delegates its authority to other servers, keeping references to those servers. There are several root servers, each covering the whole domain name space. The root servers are distributed all around the world.

Primary and Secondary Servers

DNS defines two types of servers: primary and secondary. A *primary server* is a server that stores a file about the zone for which it is an authority. It is responsible for creating, maintaining, and updating the zone file. It stores the zone file on a local disk.

A *secondary server* is a server that transfers the complete information about a zone from another server (primary or secondary) and stores the file on its local disk. The secondary server neither creates nor updates the zone files. If updating is required, it must be done by the primary server, which sends the updated version to the secondary.

The primary and secondary servers are both authoritative for the zones they serve. The idea is not to put the secondary server at a lower level of authority but to create redundancy for the data so that if one server fails, the other can continue serving clients. Note also that a server can be a primary server for a specific zone and a secondary server for another zone. Therefore, when we refer to a server as a primary or secondary server, we should be careful about which zone we refer to.

A primary server loads all information from the disk file; the secondary server loads all information from the primary server.

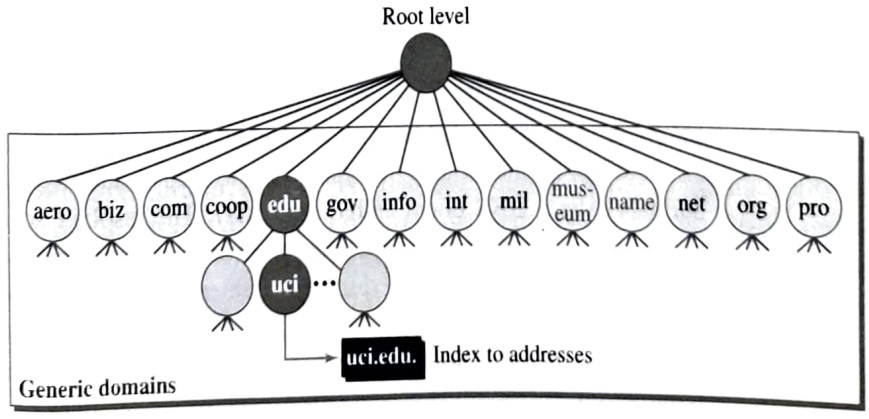
26.6.2 DNS in the Internet

DNS is a protocol that can be used in different platforms. In the Internet, the domain name space (tree) was originally divided into three different sections: generic domains, country domains, and the inverse domains. However, due to the rapid growth of the Internet, it became extremely difficult to keep track of the inverse domains, which could be used to find the name of a host when given the IP address. The inverse domains are now deprecated (see RFC 3425). We, therefore, concentrate on the first two.

Generic Domains

The **generic domains** define registered hosts according to their generic behavior. Each node in the tree defines a domain, which is an index to the domain name space database (see Figure 26.34).

Figure 26.34 *Generic domains*



Looking at the tree, we see that the first level in the **generic domains** section allows 14 possible labels. These labels describe the organization types as listed in Table 26.12.

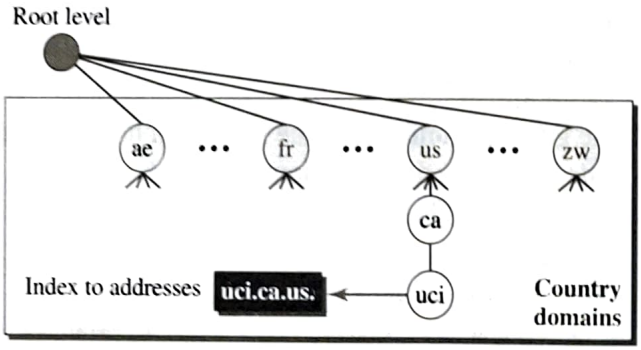
Table 26.12 *Generic domain labels*

Label	*Description*	*Label*	*Description*
aero	Airlines and aerospace	**int**	International organizations
biz	Businesses or firms	**mil**	Military groups
com	Commercial organizations	**museum**	Museums
coop	Cooperative organizations	**name**	Personal names (individuals)
edu	Educational institutions	**net**	Network support centers
gov	Government institutions	**org**	Nonprofit organizations
info	Information service providers	**pro**	Professional organizations

Country Domains

The **country domains** section uses two-character country abbreviations (e.g., us for United States). Second labels can be organizational, or they can be more specific national designations. The United States, for example, uses state abbreviations as a subdivision of us (e.g., ca.us.). Figure 26.35 shows the country domains section. The address ***uci.ca.us.*** can be translated to University of California, Irvine, in the state of California in the United States.

Figure 26.35 *Country domains*



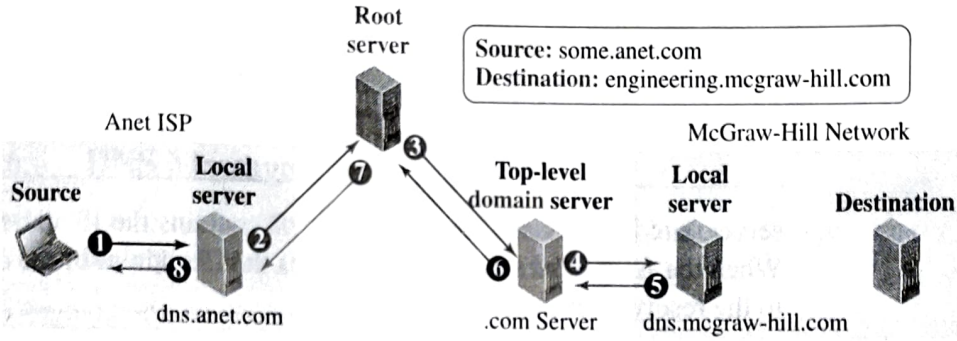
26.6.3 Resolution

Mapping a name to an address is called *name-address resolution*. DNS is designed as a client-server application. A host that needs to map an address to a name or a name to an address calls a DNS client called a ***resolver.*** The resolver accesses the closest DNS server with a mapping request. If the server has the information, it satisfies the resolver; otherwise, it either refers the resolver to other servers or asks other servers to provide the information. After the resolver receives the mapping, it interprets the response to see if it is a real resolution or an error, and finally delivers the result to the process that requested it. A resolution can be either recursive or iterative.

Recursive Resolution

Figure 26.36 shows a simple example of a **recursive resolution.** We assume that an application program running on a host named ***some.anet.com*** needs to find the IP address of another host named ***engineering.mcgraw-hill.com*** to send a message to. The source host is connected to the Anet ISP; the destination host is connected to the McGraw-Hill network.

Figure 26.36 *Recursive resolution*

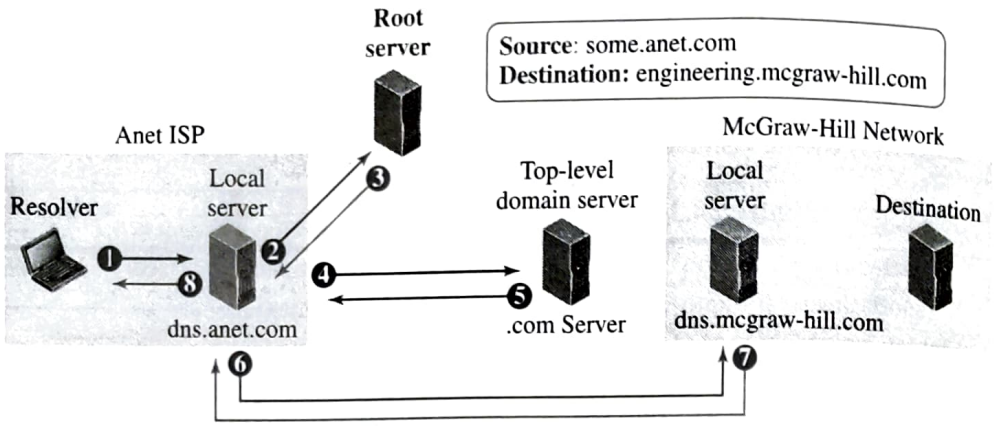


The application program on the source host calls the DNS resolver (client) to find the IP address of the destination host. The resolver, which does not know this address, sends the query to the local DNS server (for example, *dns.anet.com*) running at the Anet ISP site (event 1). We assume that this server does not know the IP address of the destination host either. It sends the query to a root DNS server, whose IP address is supposed to be known to this local DNS server (event 2). Root servers do not normally keep the mapping between names and IP addresses, but a root server should at least know about one server at each top level domain (in this case, a server responsible for *com* domain). The query is sent to this top-level-domain server (event 3). We assume that this server does not know the name-address mapping of this specific destination, but it knows the IP address of the local DNS server in the McGraw-Hill company (for example, *dns.mcgraw-hill.com*). The query is sent to this server (event 4), which knows the IP address of the destination host. The IP address is now sent back to the top-level DNS server (event 5), then back to the root server (event 6), then back to the ISP DNS server, which may cache it for the future queries (event 7), and finally back to the source host (event 8).

Iterative Resolution

In **iterative resolution,** each server that does not know the mapping sends the IP address of the next server back to the one that requested it. Figure 26.37 shows the flow of information in an iterative resolution in the same scenario as the one depicted in Figure 26.36. Normally the iterative resolution takes place between two local servers; the original resolver gets the final answer from the local server. Note that the messages shown by events 2, 4, and 6 contain the same query. However, the message shown by event 3 contains the IP address of the top-level domain server, the message shown by event 5 contains the IP address of the McGraw-Hill local DNS

Figure 26.37 *Iterative resolution*



server, and the message shown by event 7 contains the IP address of the destination. When the Anet local DNS server receives the IP address of the destination, it sends it to the resolver (event 8).

26.6.4 Caching

Each time a server receives a query for a name that is not in its domain, it needs to search its database for a server IP address. Reduction of this search time would increase efficiency. DNS handles this with a mechanism called *caching*. When a server asks for a mapping from another server and receives the response, it stores this information in its cache memory before sending it to the client. If the same or another client asks for the same mapping, it can check its cache memory and resolve the problem. However, to inform the client that the response is coming from the cache memory and not from an authoritative source, the server marks the response as *unauthoritative*.

Caching speeds up resolution, but it can also be problematic. If a server caches a mapping for a long time, it may send an outdated mapping to the client. To counter this, two techniques are used. First, the authoritative server always adds information to the mapping called *time to live* (TTL). It defines the time in seconds that the receiving server can cache the information. After that time, the mapping is invalid and any query must be sent again to the authoritative server. Second, DNS requires that each server keep a TTL counter for each mapping it caches. The cache memory must be searched periodically and those mappings with an expired TTL must be purged.

26.6.5 Resource Records

The zone information associated with a server is implemented as a set of *resource records*. In other words, a name server stores a database of resource records. A *resource record* is a 5-tuple structure, as shown below:

(Domain Name, Type, Class, TTL, Value)

The domain name field is what identifies the resource record. The value defines the information kept about the domain name. The TTL defines the number of

seconds for which the information is valid. The class defines the type of network; we are only interested in the class IN (Internet). The type defines how the value should be interpreted. Table 26.13 lists the common types and how the value is interpreted for each type.

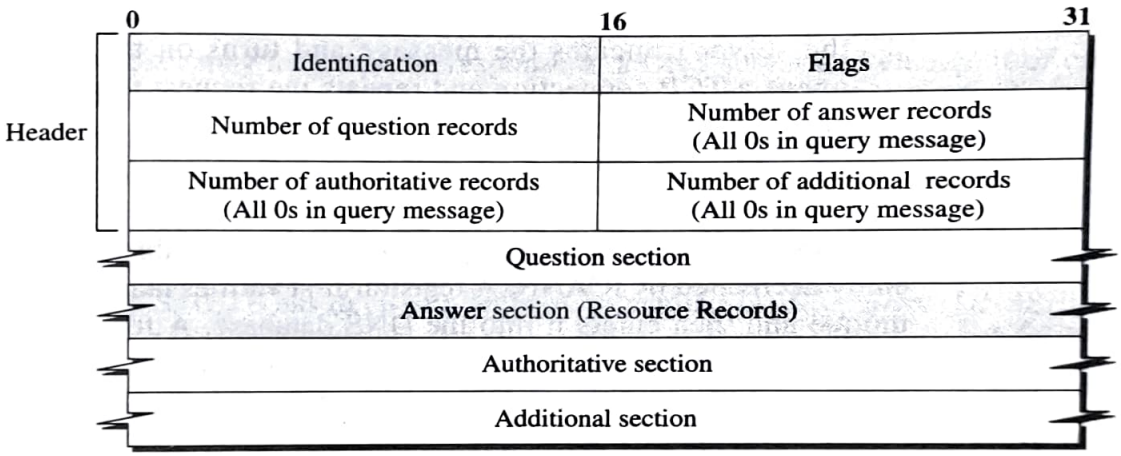
Table 26.13 *Types*

Type	*Interpretation of value*
A	A 32-bit IPv4 address (see Chapter 18)
NS	Identifies the authoritative servers for a zone
CNAME	Defines an alias for the official name of a host
SOA	Marks the beginning of a zone
MX	Redirects mail to a mail server
AAAA	An IPv6 address (see Chapter 22)

26.6.6 DNS Messages

To retrieve information about hosts, DNS uses two types of messages: *query* and *response*. Both types have the same format as shown in Figure 26.38.

Figure 26.38 *DNS message*



Note:
The query message contains only the question section.
The response message includes the question section, the answer section, and possibly two other sections.

We briefly discuss the fields in a DNS message. The identification field is used by the client to match the response with the query. The flag field defines whether the message is a query or response. It also includes status of error. The next four fields in the header define the number of each record type in the message. The question section consists of one or more question records. It is present in both query and response messages. The answer section consists of one or more resource records. It is present only in response messages. The authoritative section gives information (domain name) about one or more authoritative servers for the query. The additional information section provides additional information that may help the resolver.

Example 26.13

In UNIX and Windows, the *nslookup* utility can be used to retrieve address/name mapping. The following shows how we can retrieve an address when the domain name is given.

```
$nslookup www.forouzan.biz
Name: www.forouzan.biz
Address: 198.170.240.179
```

Encapsulation

DNS can use either UDP or TCP. In both cases the well-known port used by the server is port 53. UDP is used when the size of the response message is less than 512 bytes because most UDP packages have a 512-byte packet size limit. If the size of the response message is more than 512 bytes, a TCP connection is used. In that case, one of two scenarios can occur:

- If the resolver has prior knowledge that the size of the response message is more than 512 bytes, it uses the TCP connection. For example, if a secondary name server (acting as a client) needs a zone transfer from a primary server, it uses the TCP connection because the size of the information being transferred usually exceeds 512 bytes.
- If the resolver does not know the size of the response message, it can use the UDP port. However, if the size of the response message is more than 512 bytes, the server truncates the message and turns on the TC bit. The resolver now opens a TCP connection and repeats the request to get a full response from the server.

26.6.7 Registrars

How are new domains added to DNS? This is done through a *registrar,* a commercial entity accredited by ICANN. A registrar first verifies that the requested domain name is unique and then enters it into the DNS database. A fee is charged. Today, there are many registrars; their names and addresses can be found at

http://www.intenic.net

To register, the organization needs to give the name of its server and the IP address of the server. For example, a new commercial organization named *wonderful* with a server named *ws* and IP address 200.200.200.5 needs to give the following information to one of the registrars:

Domain name: ws.wonderful.com **IP address:** 200.200.200.5

26.6.8 DDNS

When the DNS was designed, no one predicted that there would be so many address changes. In DNS, when there is a change, such as adding a new host, removing a host, or changing an IP address, the change must be made to the DNS master file. These types of changes involve a lot of manual updating. The size of today's Internet does not allow for this kind of manual operation.

The DNS master file must be updated dynamically. The **Dynamic Domain Name System (DDNS)** therefore was devised to respond to this need. In DDNS, when a binding between a name and an address is determined, the information is sent, usually by DHCP (discussed in Chapter 18) to a primary DNS server. The primary server updates the zone. The secondary servers are notified either actively or passively. In active notification, the primary server sends a message to the secondary servers about the change in the zone, whereas in passive notification, the secondary servers periodically check for any changes. In either case, after being notified about the change, the secondary server requests information about the entire zone (called the *zone transfer*).

To provide security and prevent unauthorized changes in the DNS records, DDNS can use an authentication mechanism.

26.6.9 Security of DNS

DNS is one of the most important systems in the Internet infrastructure; it provides crucial services to Internet users. Applications such as Web access or e-mail are heavily dependent on the proper operation of DNS. DNS can be attacked in several ways including:

1. The attacker may read the response of a DNS server to find the nature or names of sites the user mostly accesses. This type of information can be used to find the user's profile. To prevent this attack, DNS messages need to be confidential (see Chapters 31 and 32).
2. The attacker may intercept the response of a DNS server and change it or create a totally new bogus response to direct the user to the site or domain the attacker wishes the user to access. This type of attack can be prevented using message origin authentication and message integrity (see Chapters 31 and 32).
3. The attacker may flood the DNS server to overwhelm it or eventually crash it. This type of attack can be prevented using the provision against denial-of-service attack.

To protect DNS, IETF has devised a technology named *DNS Security* (DNSSEC) that provides message origin authentication and message integrity using a security service called *digital signature* (see Chapter 31). DNSSEC, however, does not provide confidentiality for the DNS messages. There is no specific protection against the denial-of-service attack in the specification of DNSSEC. However, the caching system protects the upper-level servers against this attack to some extent.

26.7 END-CHAPTER MATERIALS

26.7.1 Recommended Reading

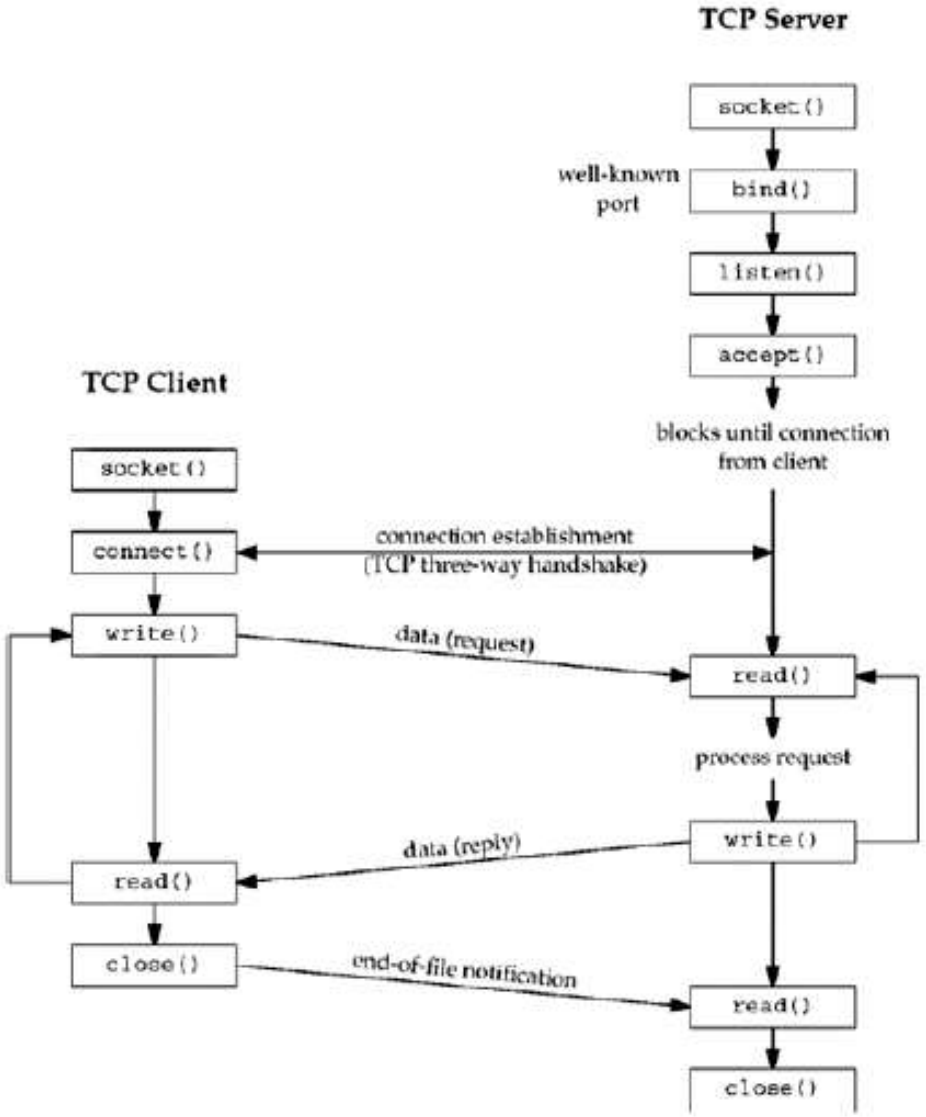
For more details about subjects discussed in this chapter, we recommend the following books and RFCs. The items enclosed in brackets refer to the reference list at the end of the book.

Books

Several books give thorough coverage of materials discussed in this chapter including [Com 06], [Mir 07], [Ste 94], [Tan 03], [Bar et al. 05].

Elementary TCP Sockets

Socket functions for elementary TCP client/server



socket Function

```
#include <sys/socket.h>

int socket (int family, int type, int protocol);

                                  Returns: non-negative descriptor if OK, -1 on error
```

- ***family*** specifies the protocol family and is one of the constants shown in Figure

family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 15)
AF_ROUTE	Routing sockets (Chapter 18)
AF_KEY	Key socket (Chapter 19)

type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

- The socket ***type*** is one of the constants shown in Figure
- The ***protocol*** argument to the socket function should be set to the specific protocol type found in Figure, or 0 to select the system's default for the given combination of *family* and *type*

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

AF_*xxx* Versus PF_*xxx*

- The "AF_" prefix stands for "address family"
- The "PF_" prefix stands for "protocol family“
- Historically, the intent was that a single protocol family might support multiple address families
- The PF_ value was used to create the socket
- The AF_ value was used in socket address structures
- But in actuality, a protocol family supporting multiple address families has never been supported
- The <sys/socket.h> header defines the PF_ value for a given protocol to be equal to the AF_ value for that protocol
- While there is no guarantee that this equality between the two will always be true

connect Function

- The connect function is used by a TCP client to establish a connection with a TCP server
- *sockfd* is a socket descriptor returned by the socket function
- The second and third arguments are a pointer to a socket address structure and its size
- The socket address structure must contain the IP address and port number of the server

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

Returns: 0 if OK, -1 on error

bind Function

- The bind function assigns a local protocol address to a socket (*sockfd)*
- With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number
- The second argument is a pointer to a protocol-specific address
- The third argument is the size of this address structure

```
#include <sys/socket.h>
int bind (intsockfd,const struct sockaddr *myaddr,socklen_taddrlen);
                                                    Returns: 0 if OK,-1 on error
```

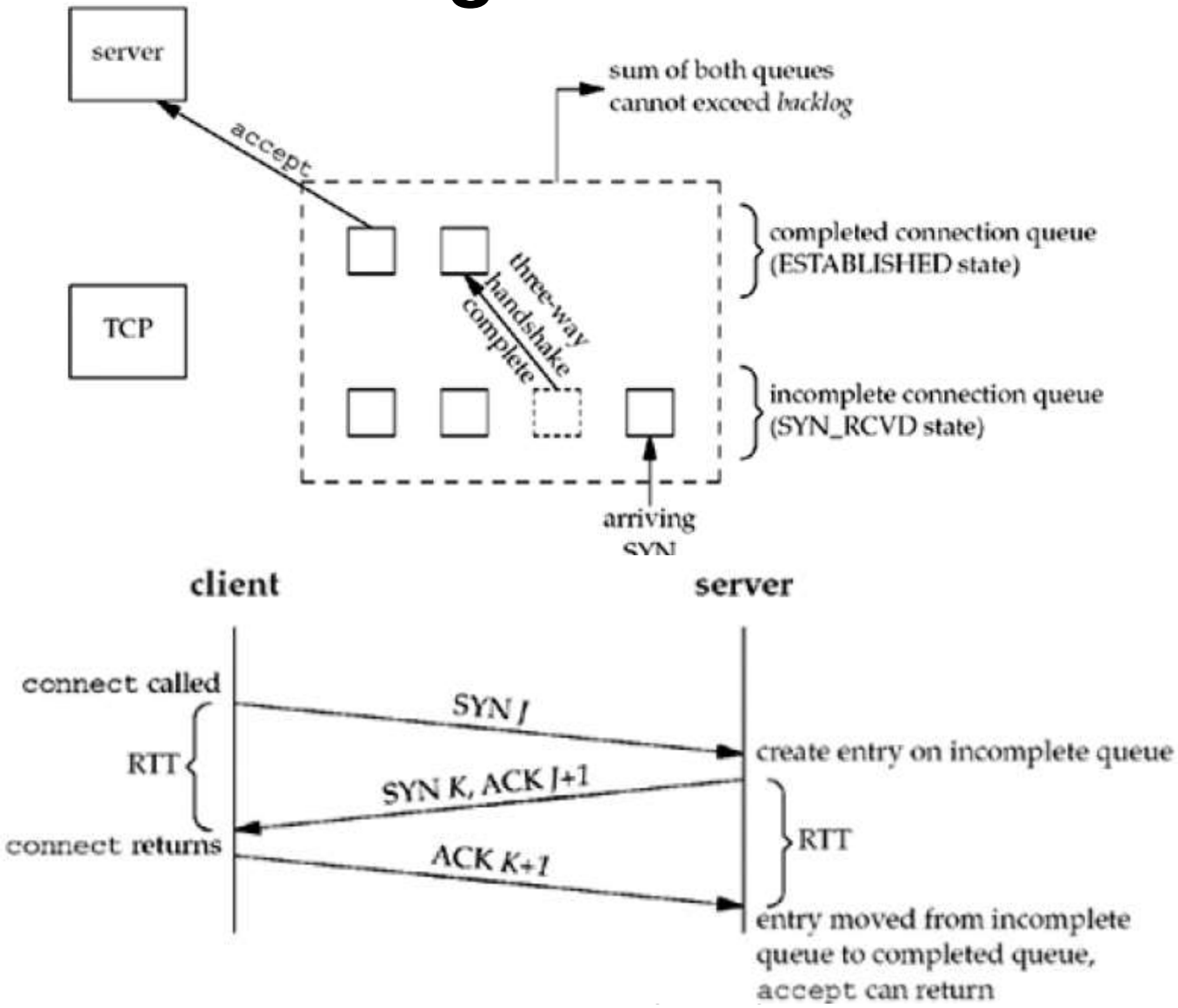
listen Function

- The listen function is called only by a TCP server and it performs two actions:
- When a socket is created by the socket function, it is assumed to be an active socket, that is, a client socket that will issue a connect
- The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket
- The second argument to this function specifies the maximum number of connections the kernel should queue for this socket

```
#include <sys/socket.h>
#int listen (intsockfd,intbacklog);
```

Returns: 0 if OK, -1 on error

The two queues maintained by TCP for a listening socket



accept Function

- accept is called by a TCP server to return the next completed connection from the front of the completed connection queue
- If the completed connection queue is empty, the process is put to sleep (assuming the default of a blocking socket)

```
#include <sys/socket.h>
int accept (intsockfd,struct sockaddr *cliaddr,socklen_t *addrlen);
```

Returns: non-negative descriptor if OK, -1 on error

- The *cliaddr* and *addrlen* arguments are used to return the protocol address of the connected peer process (the client)
- *addrlen* is referred by the integer value to the size of the socket address structure pointed to by *cliaddr*
- If accept is successful, its return value is a brand-new descriptor automatically created by the kernel that refers to the TCP connection with the client

Daytime server that prints client IP address and port

- See Code from: *intro/daytimetcpsrv1.c*

solaris % **daytimetcpcli 127.0.0.1**
Thu Sep 11 12:44:00 2003
solaris % **daytimetcpcli 192.168.1.20**
Thu Sep 11 12:44:09 2003

- We first specify the server's IP address as the loopback address (127.0.0.1) then as its own IP address (192.168.1.20)
- Here is the corresponding server output:

solaris # **daytimetcpsrv1**
connection from 127.0.0.1, port 43388
connection from 192.168.1.20, port 43389

close Function

- The normal Unix close function is also used to close a socket and terminate a TCP connection

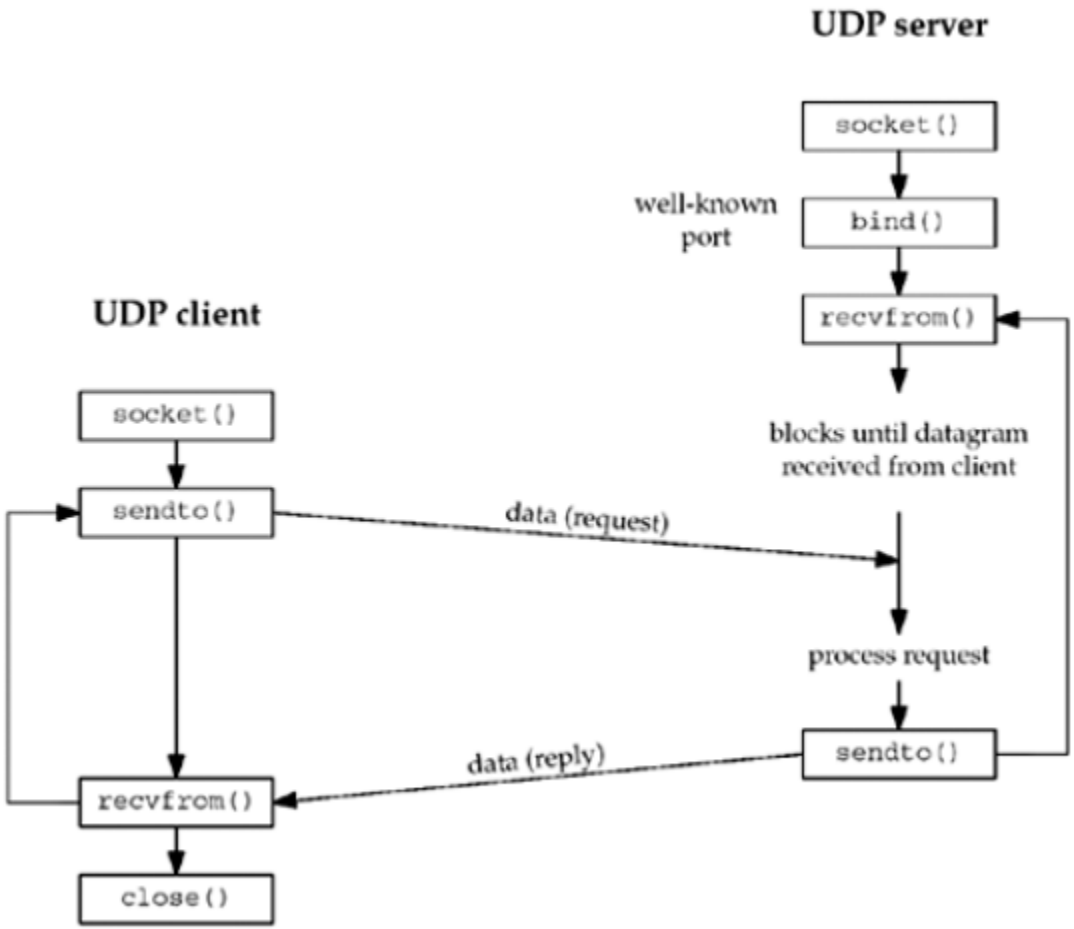
```
#include <unistd.h>
int close (int sockfd);
```

Returns: 0 if OK, -1 on error

Thank you

Elementary UDP Sockets

Socket functions for UDP client/server



recvfrom and *sendto* Functions

- These two functions are similar to the standard read and write functions, but three additional arguments are required

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct sockaddr
*from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const struct
sockaddr *to, socklen_t addrlen);
```

Both return: number of bytes read or written if OK, −1 on error

- The first three arguments, *sockfd, buff*, and *nbytes*, are identical to the first three arguments for read and write: descriptor, pointer to buffer to read into or write from, and number of bytes to read or write

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct sockaddr
*from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const struct
sockaddr *to, socklen_t addrlen);
```

Both return: number of bytes read or written if OK, –1 on error

- For sendto: a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is to be sent
- The size of this socket address structure is specified by *addrlen*
- For recvfrom: function fills in the socket address structure pointed to by *from* with the protocol address of who sent the datagram
- The number of bytes stored in this socket address structure is also returned to the caller in the integer pointed to by *addrlen*
- For now, we will always set the *flags* to 0

Simple echo client/server using UDP

UDP echo server

```
#include "unp.h"

int main(int argc, char **argv)
{
    int sockfd;
    struct sockaddr_in servaddr, cliaddr;
    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);
    Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));
    dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
}
```

dg_echo Function

```
#include "unp.h"
Void dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
{
    int n;
    socklen_t len;
    char mesg[MAXLINE];
    for ( ; ; ) {
        len = clilen;
        n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
        Sendto(sockfd, mesg, n, 0, pcliaddr, len);
    }
}
```

UDP Echo Client: main Function

```
#include "unp.h"
Int main(int argc, char **argv)
{
    int sockfd;
    struct sockaddr_in servaddr;
    if(argc != 2)
        err_quit("usage: udpcli <IPaddress>");
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
    dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));
    exit(0);
}
```

UDP Echo Client: dg_cli Function

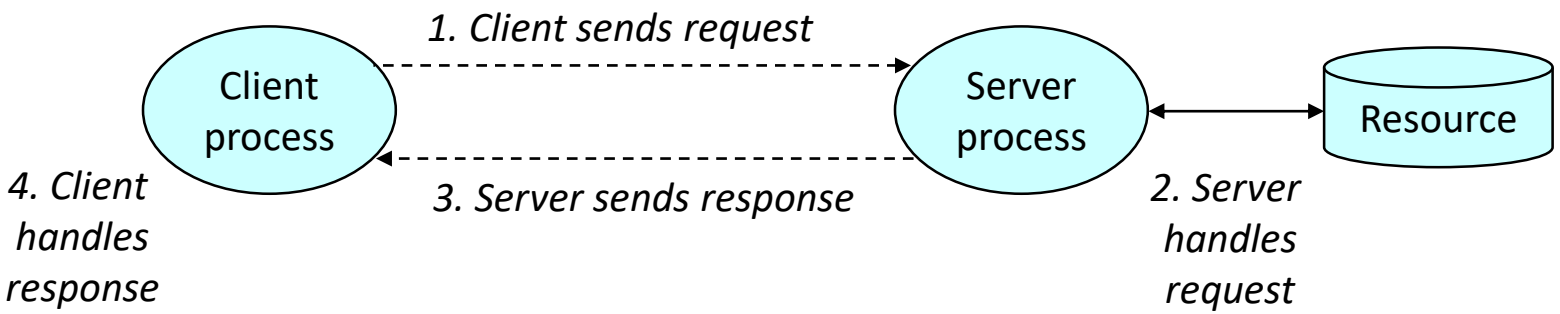
```
#include "unp.h"
void dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int n;
    char sendline[MAXLINE], recvline[MAXLINE + 1];
    while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
        n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
        recvline[n] = 0; /* null terminate */
        Fputs(recvline, stdout);
    }
}
```

Thank you

Programmer’s view of the Internet

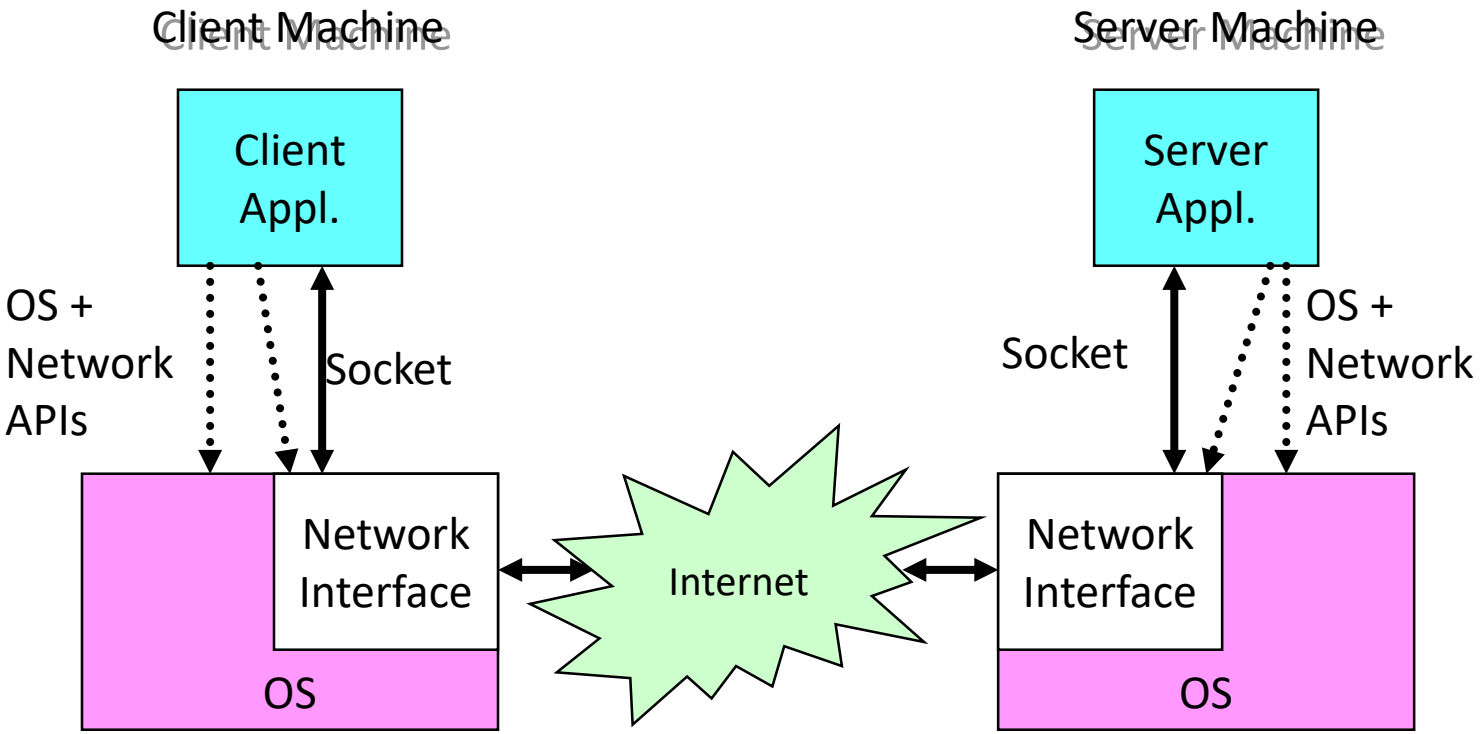
A Client-Server Exchange

- A *server* process and one or more *client* processes
- Server manages some *resource*.
- Server provides *service* by manipulating resource for clients.



Note: clients and servers are processes running on hosts (can be the same or different hosts).

Network Applications



Access to Network via Program Interface

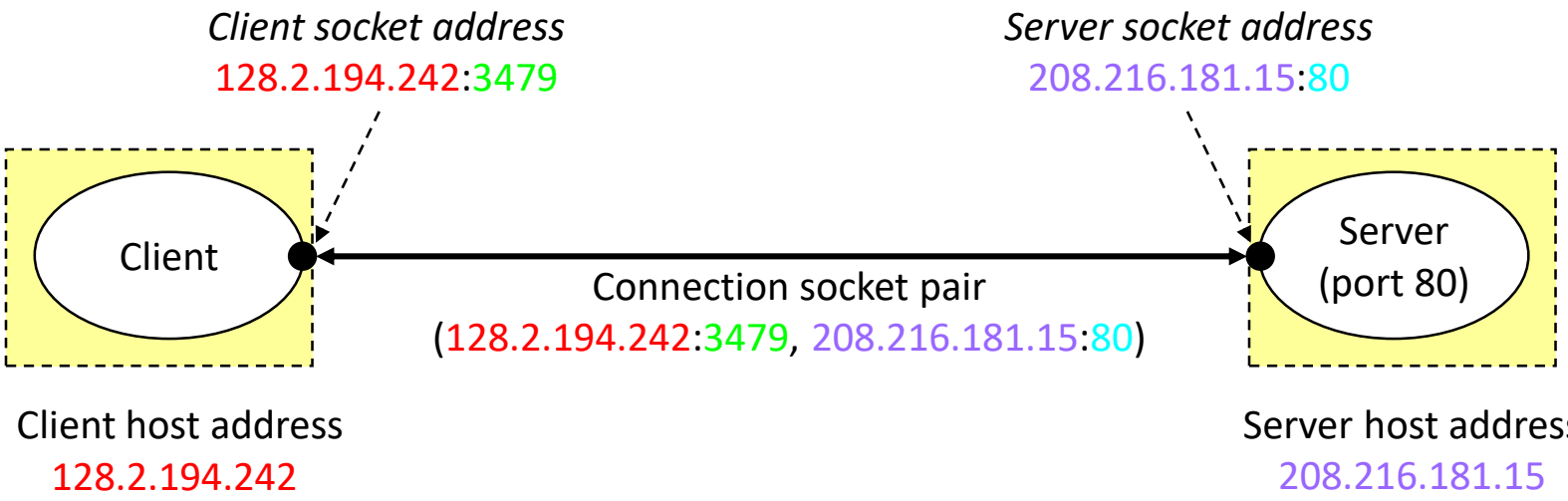
- Sockets make network I/O look like files
- Call system functions to control and communicate
- Network code handles issues of routing, segmentation.

Internet Connections (TCP/IP)

Two common paradigms for clients and servers communication

- Datagram Socket(UDP protocol SOCK_DGRAM)
- Stream Socket (TCP protocol, SOCK_STREAM)

Connections are point-to-point, full-duplex (2-way communication), and reliable.



Clients

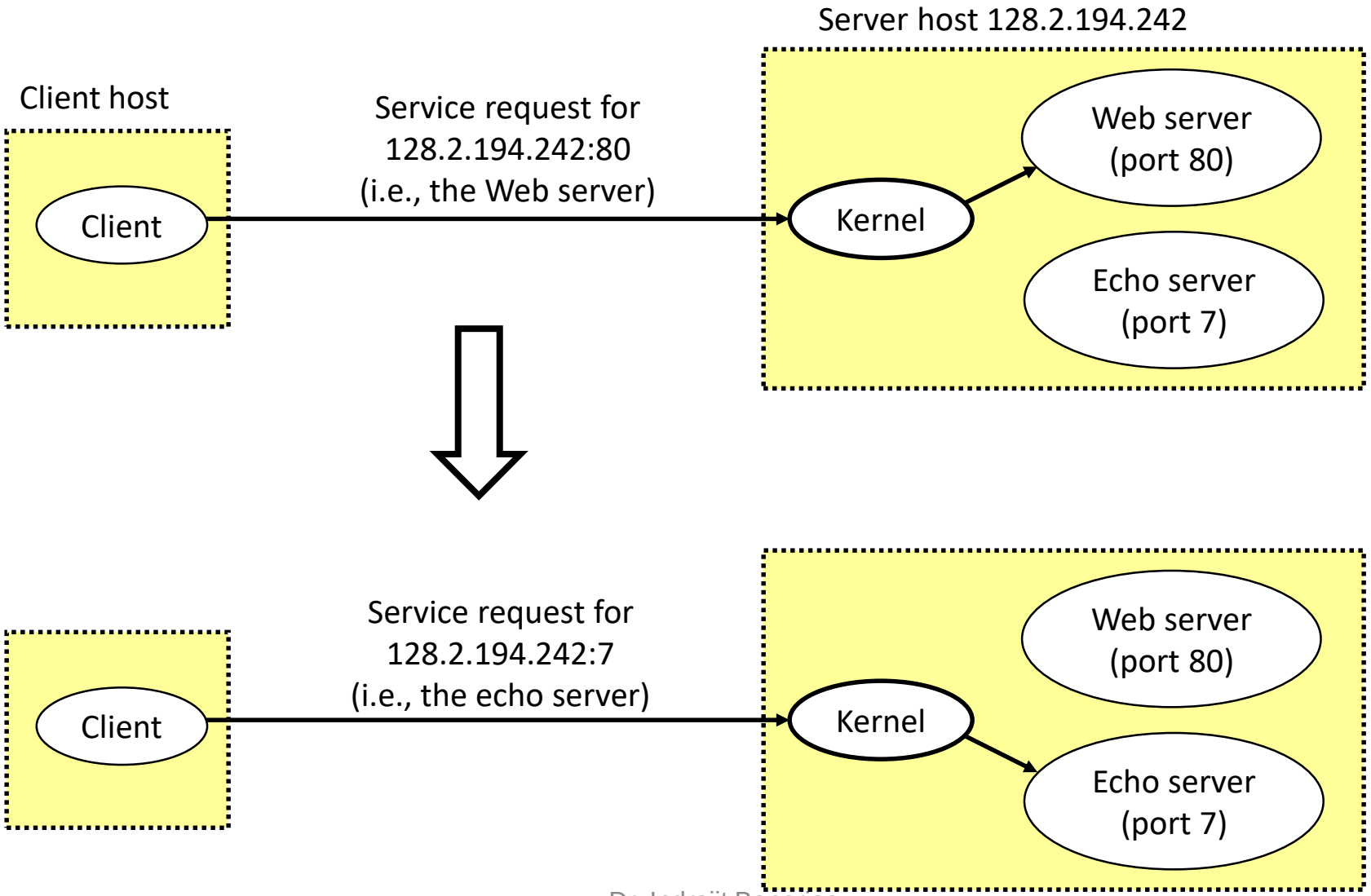
Examples of client programs

- Web browsers, `ftp`, `telnet`, `ssh`

How does a client find the server?

- The IP address in the server socket address identifies the host *(more precisely, an adaptor on the host)*
- The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.
- Examples of well known ports
  - Port 7: Echo server
  - Port 23: Telnet server
  - Port 25: Mail server
  - Port 80: Web server

Using Ports to Identify Services



Servers

- Servers are long-running processes (daemons).
  - Created at boot-time (typically) by the init process (process 1)
  - Run continuously until the machine is turned off.
- Each server waits for requests to arrive on a well-known port associated with a particular service.
  - Port 7: echo server
  - Port 23: telnet server
  - Port 25: mail server
  - Port 80: HTTP server
- A machine that runs a server process is also often referred to as a “server.”

See `/etc/services` for a comprehensive list of the services available on a Linux machine.

Sockets Introduction

Socket Address Structures

- An IPv4 socket address structure, commonly called an "Internet socket address structure,"
- It is named sockaddr_in and is defined by including the <netinet/in.h> header

```
struct in_addr {
  in_addr_t    s_addr;           /* 32-bit IPv4 address */
                                 /* network byte ordered */
};

struct sockaddr_in {
  uint8_t          sin_len;      /* length of structure (16) */
  sa_family_t      sin_family;   /* AF_INET */
  in_port_t        sin_port;     /* 16-bit TCP or UDP port number */
                                 /* network byte ordered */
  struct in_addr   sin_addr;     /* 32-bit IPv4 address */
                                 /* network byte ordered */
  char             sin_zero[8];  /* unused */
};
```

Datatypes required by the POSIX specification

Datatype	Description	Header
int8_t	Signed 8-bit integer	<sys/types.h>
uint8_t	Unsigned 8-bit integer	<sys/types.h>
int16_t	Signed 16-bit integer	<sys/types.h>
uint16_t	Unsigned 16-bit integer	<sys/types.h>
int32_t	Signed 32-bit integer	<sys/types.h>
uint32_t	Unsigned 32-bit integer	<sys/types.h>
sa_family_t	Address family of socket address structure	<sys/socket.h>
socklen_t	Length of socket address structure, normally uint32_t	<sys/socket.h>
in_addr_t	IPv4 address, normally uint32_t	<netinet/in.h>
in_port_t	TCP or UDP port, normally uint16_t	<netinet/in.h>

Generic Socket Address Structure

- How to declare the type of pointer that may accept any type of pointer?
- With ANSI C, the solution is simple: void * is the generic pointer type
- The socket functions when take an argument it is defining a variable of *generic* socket address structure define in the <sys/socket.h>

```
struct sockaddr {
  uint8_t       sa_len;
  sa_family_t   sa_family;    /* address family: AF_xxx value */
  char          sa_data[14];  /* protocol-specific address */
};
```

IPv6 Socket Address Structure

- The IPv6 socket address is defined by including the <netinet/in.h> header

```
struct in6_addr {
  uint8_t  s6_addr[16];          /* 128-bit IPv6 address */
                                 /* network byte ordered */
};

#define SIN6_LEN      /* required for compile-time tests */

struct sockaddr_in6 {
  uint8_t         sin6_len;      /* length of this struct (28) */
  sa_family_t     sin6_family;   /* AF_INET6 */
  in_port_t       sin6_port;     /* transport layer port# */
                                 /* network byte ordered */
  uint32_t        sin6_flowinfo; /* flow information, undefined */
  struct in6_addr sin6_addr;     /* IPv6 address */
                                 /* network byte ordered */
  uint32_t        sin6_scope_id; /* set of interfaces for a scope */
};
```

New Generic Socket Address Structure

- A new generic socket address structure was defined as part of the IPv6 sockets API
- This is to overcome some of the shortcomings of the existing struct sockaddr
- The new struct sockaddr_storage is large enough to hold any socket address type supported by the system

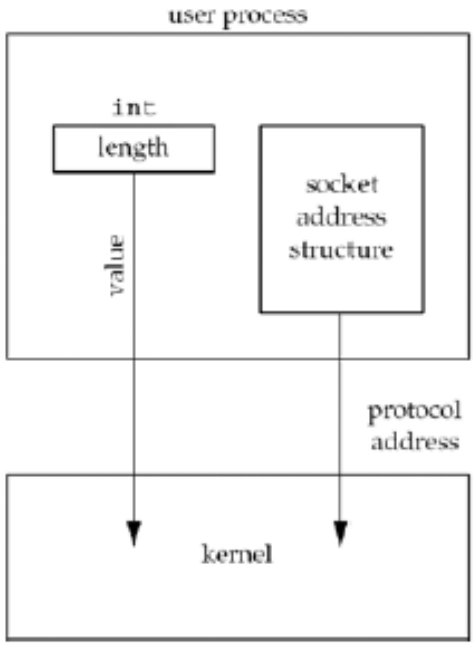
```
struct sockaddr_storage {
  uint8_t      ss_len;       /* length of this struct (implementation dependent)
  sa_family_t  ss_family;    /* address family: AF_xxx value */
  /* implementation-dependent elements to provide:
   * a) alignment sufficient to fulfill the alignment requirements of
   *    all socket address types that the system supports.
   * b) enough storage to hold any type of socket address that the
   *    system supports.
   */
};
```

Value-Result Arguments

- We mentioned that when a socket address structure is passed to any socket function, it is always passed by reference
  - That is, a pointer to the structure is passed
- The length of the structure is also passed as an argument
  - But the way in which the length is passed depends on which direction the structure is being passed:
    - from the process to the kernel
      - bind, connect, and sendto, pass a socket address structure from the process to the kernel
    - or vice versa
      - accept, recvfrom, getsockname, and getpeername, pass a socket address structure from the kernel to the process

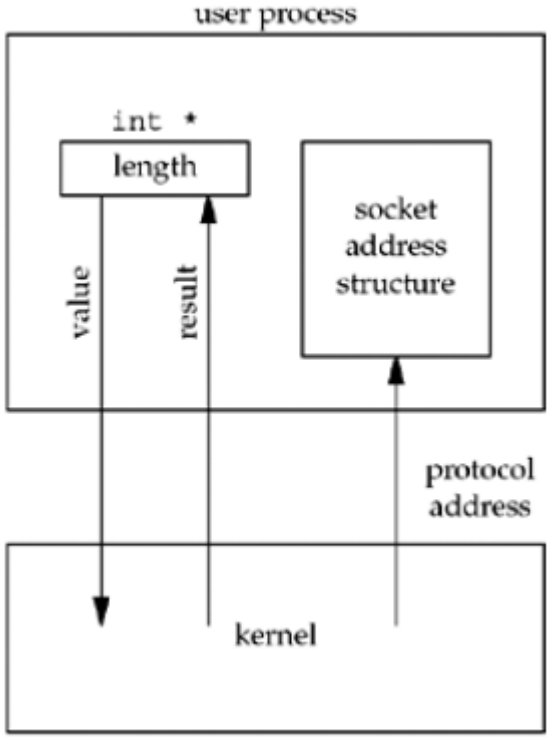
Socket address structure passed from process to kernel

```
#define    SA          struct sockaddr
struct sockaddr_in serv;

/* fill in serv{} */
connect (sockfd, (SA *) &serv, sizeof(serv));
```



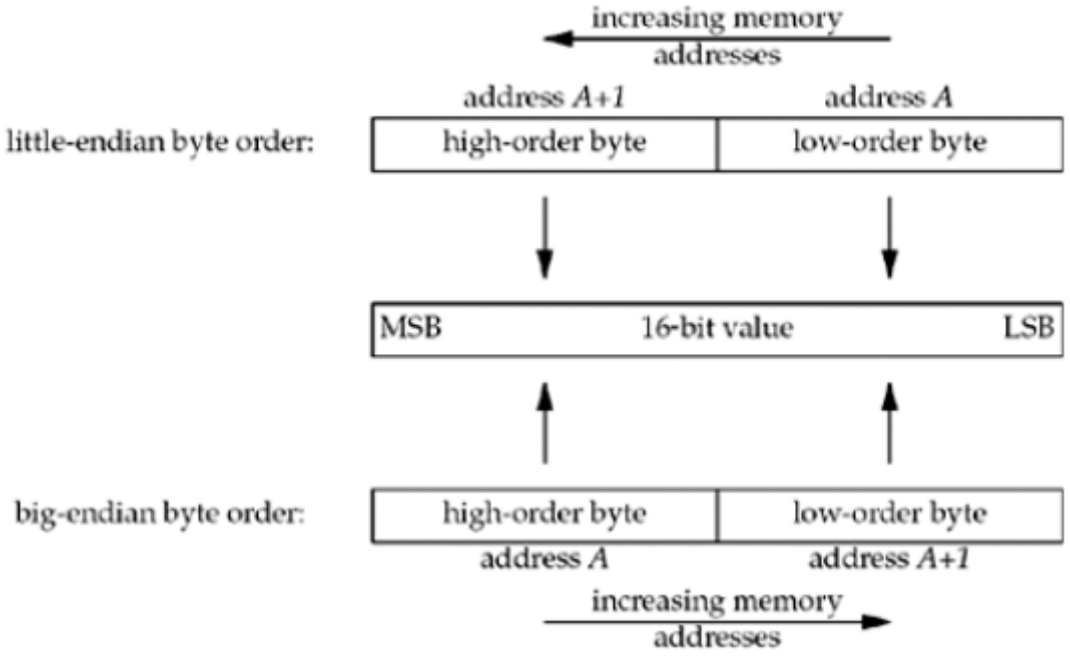
Socket address structure passed from kernel to process

```
struct sockaddr_un  cli;    /* Unix domain */
socklen_t  len;

len = sizeof(cli);          /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```



Byte Ordering Functions

- Consider a 16-bit integer that is made up of 2 bytes
- There are two ways to store the two bytes in memory:
  - with the low-order byte at the starting address, known as *little-endian* byteorder
  - with the high-order byte at the starting address, known as *big-endian* byte order



Byte Manipulation Functions

- There are two groups of functions that operate on multibyte fields, without interpreting the data, and without assuming that the data is a null-terminated C string
- We need these types of functions when dealing with socket address structures because we need to manipulate fields such as IP addresses
- The first group of functions, whose names begin with b (for byte), are from 4.2BSD and are still provided by almost any system that supports the socket functions
- The second group of functions, whose names begin with mem (for memory), are from the ANSI C standard and are provided with any system that supports an ANSI C library

Byte Manipulation Functions

```
#include <strings.h>
void bzero(void *dest,size_t nbytes);
void bcopy(const void *src,void *dest,size_t nbytes);
int bcmp(const void *ptr1,const void *ptr2,size_t nbytes);
```

Returns: 0 if equal, nonzero if unequal

```
#include <string.h>
void *memset(void *dest,int c,size_t len);
void *memcpy(void *dest,const void *src,size_t nbytes);
int memcmp(const void *ptr1,const void *ptr2,size_t nbytes);
```

Returns: 0 if equal, <0 or >0 if unequal (see text)

inet_aton, inet_addr, and inet_ntoa Functions

- inet_aton, inet_ntoa, and inet_addr convert an IPv4 address from a dotted-decimal string (e.g., "206.168.112.96") to its 32-bit network byte ordered binary value

`#include <arpa/inet.h>`
`int inet_aton(const char *strptr,struct in_addr *addrptr);`
Returns: 1 if string was valid, 0 on error
`in_addr_t inet_addr(const char *strptr);`
Returns: 32-bit binary network byte ordered IPv4 address; `INADDR_NONE` if error
`char *inet_ntoa(struct in_addrinaddr);`
Returns: pointer to dotted-decimal string

int inet_aton(const char *_strptr_, struct in_addr *_addrptr_);

- inet_aton, converts the C character string pointed to by *strptr* into its 32- bit binary network byte ordered value, which is stored through the pointer *addrptr*
- If successful, 1 is returned; otherwise, 0 is returned

in_addr_t inet_addr(const char *_strptr_);

- inet_addr does the same conversion, returning the 32-bit binary network byte ordered value as the return value

char *inet_ntoa(struct in_addr *inaddr*);

- The inet_ntoa function converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string

inet_pton and inet_ntop Functions

- These two functions work with both IPv4 and IPv6 addresses
- The letters "p" and "n" stand for *presentation (*ASCII string) and *numeric (*binary value)

```
#include <arpa/inet.h>
int inet_pton(int family, const char *strptr, void *addrptr);
```

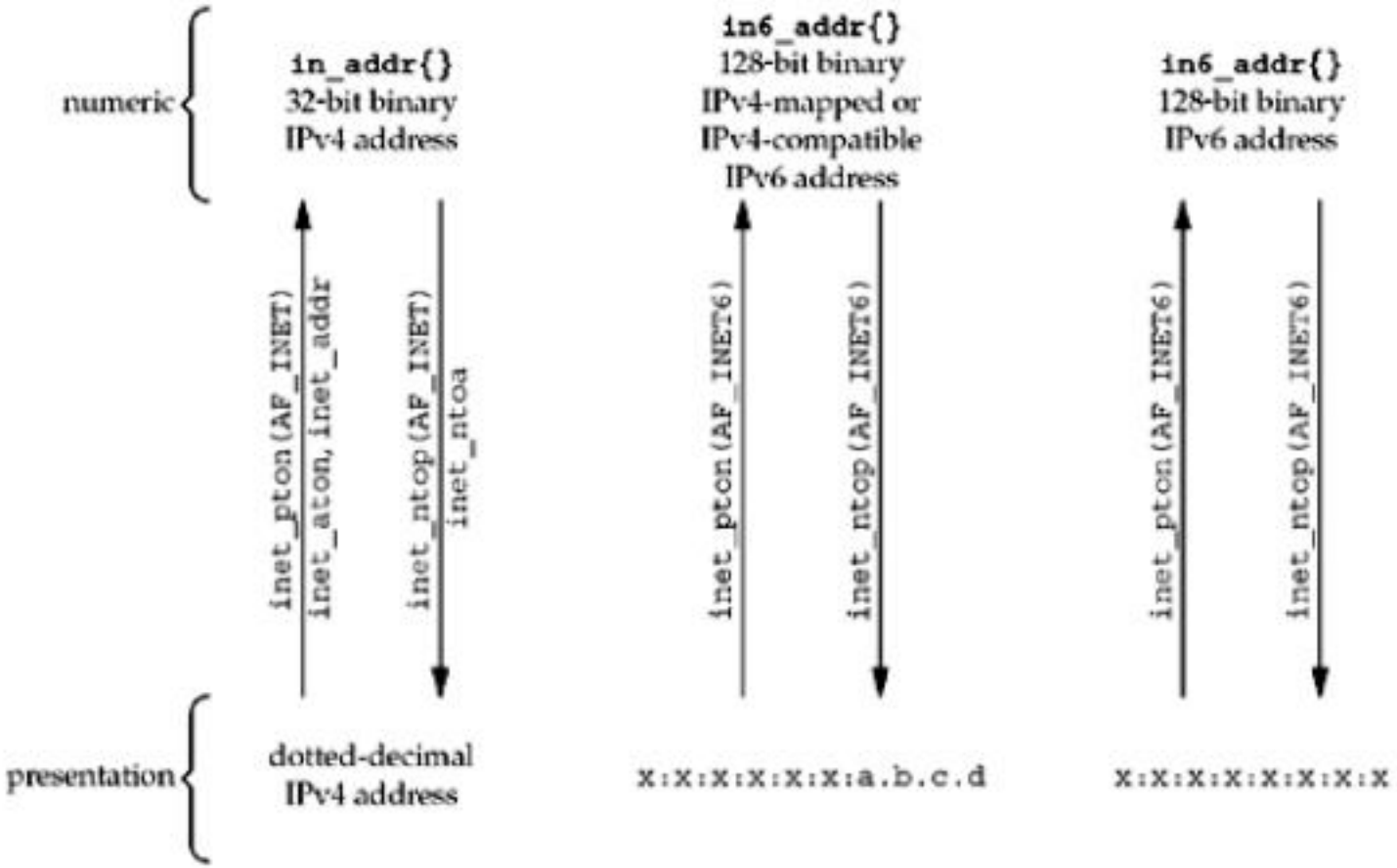
Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error

```
const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);
```

Returns: pointer to result if OK, NULL on error

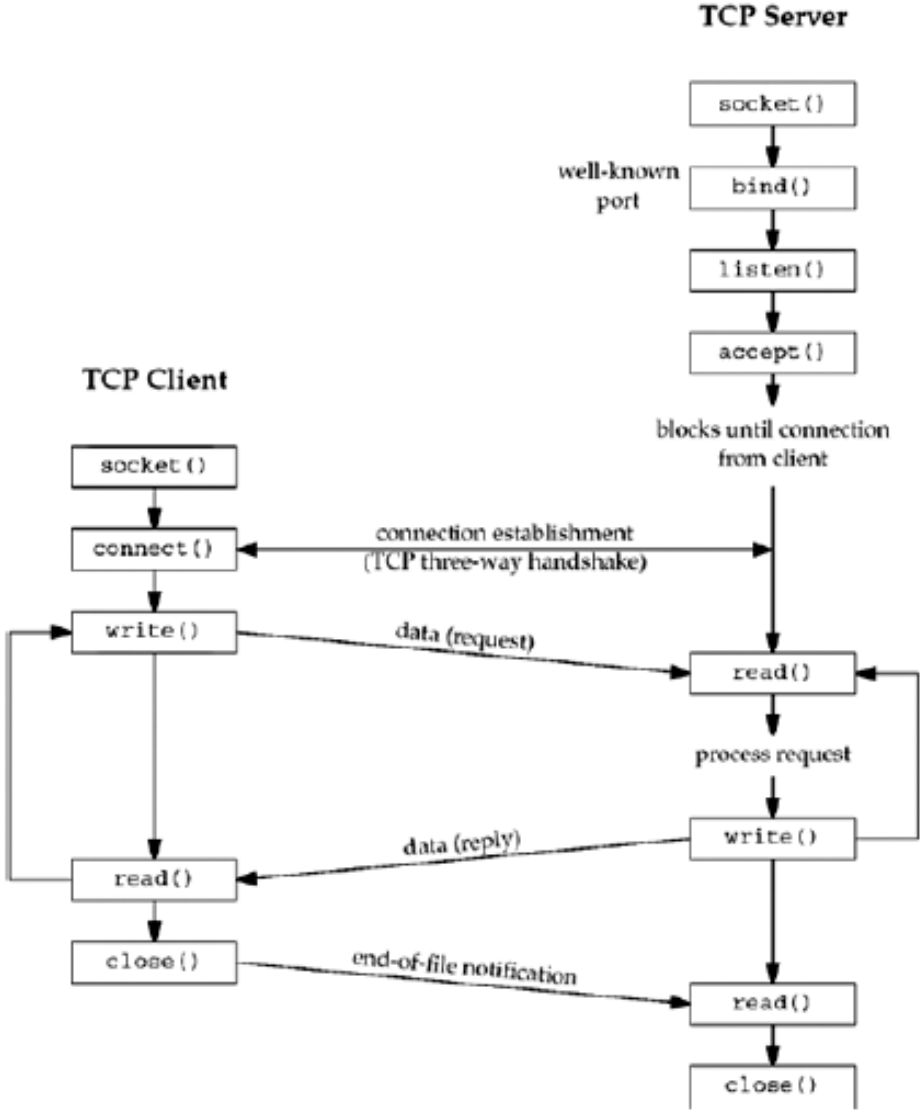
- The *family* argument for both functions is either AF_INET or AF_INET6

Summary of address conversion functions



Elementary TCP Sockets

Socket functions for elementary TCP client/server

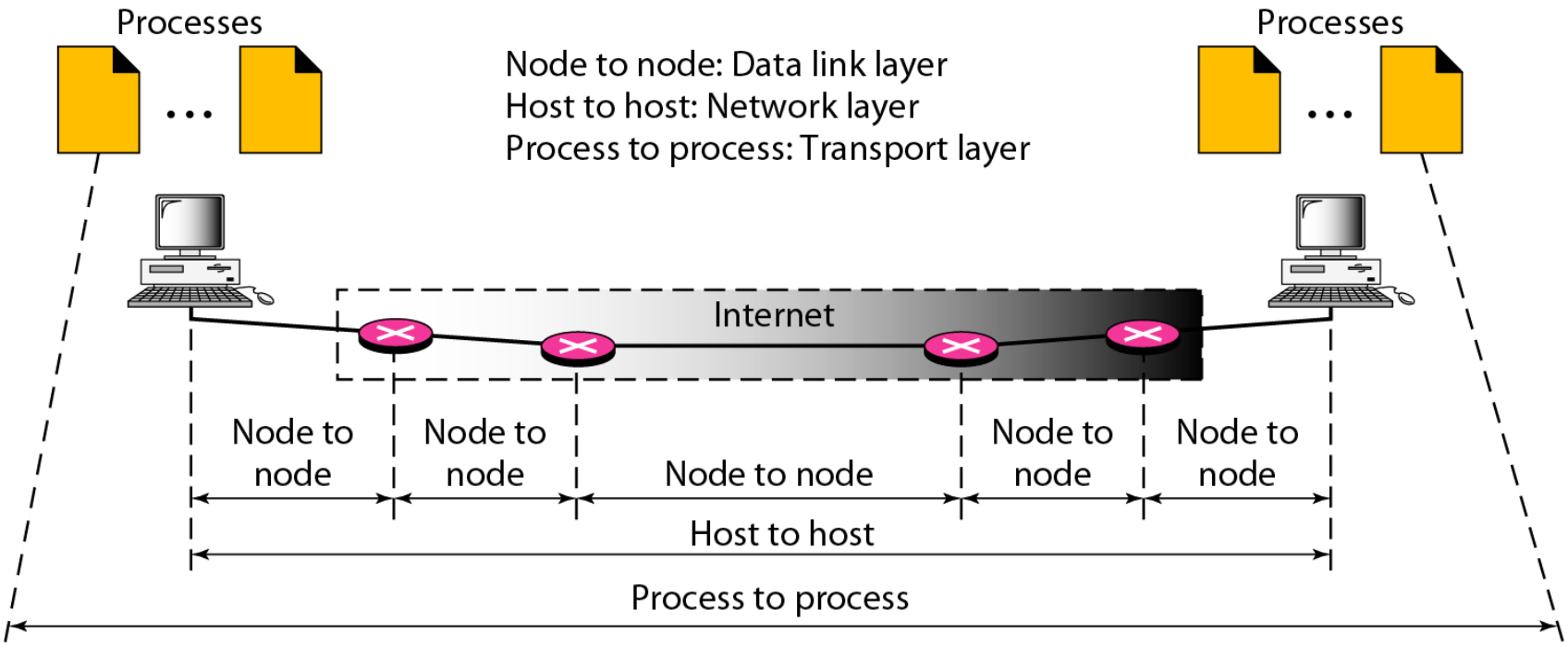


The Transport Layer: TCP, UDP

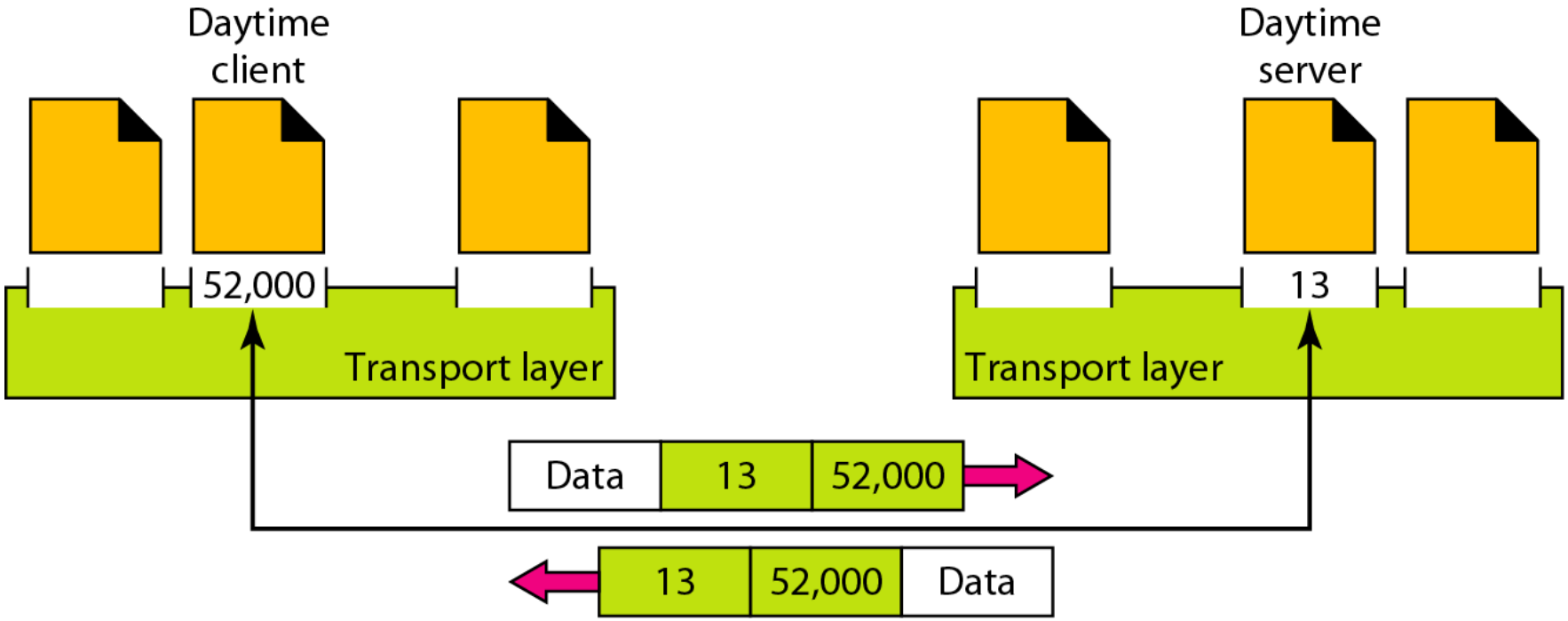
PROCESS-TO-PROCESS DELIVERY

- *The transport layer is responsible for process-to-process delivery— the delivery of a packet, part of a message, from one process to another*
- *Two processes communicate in a client/server relationship, as we will see later.*

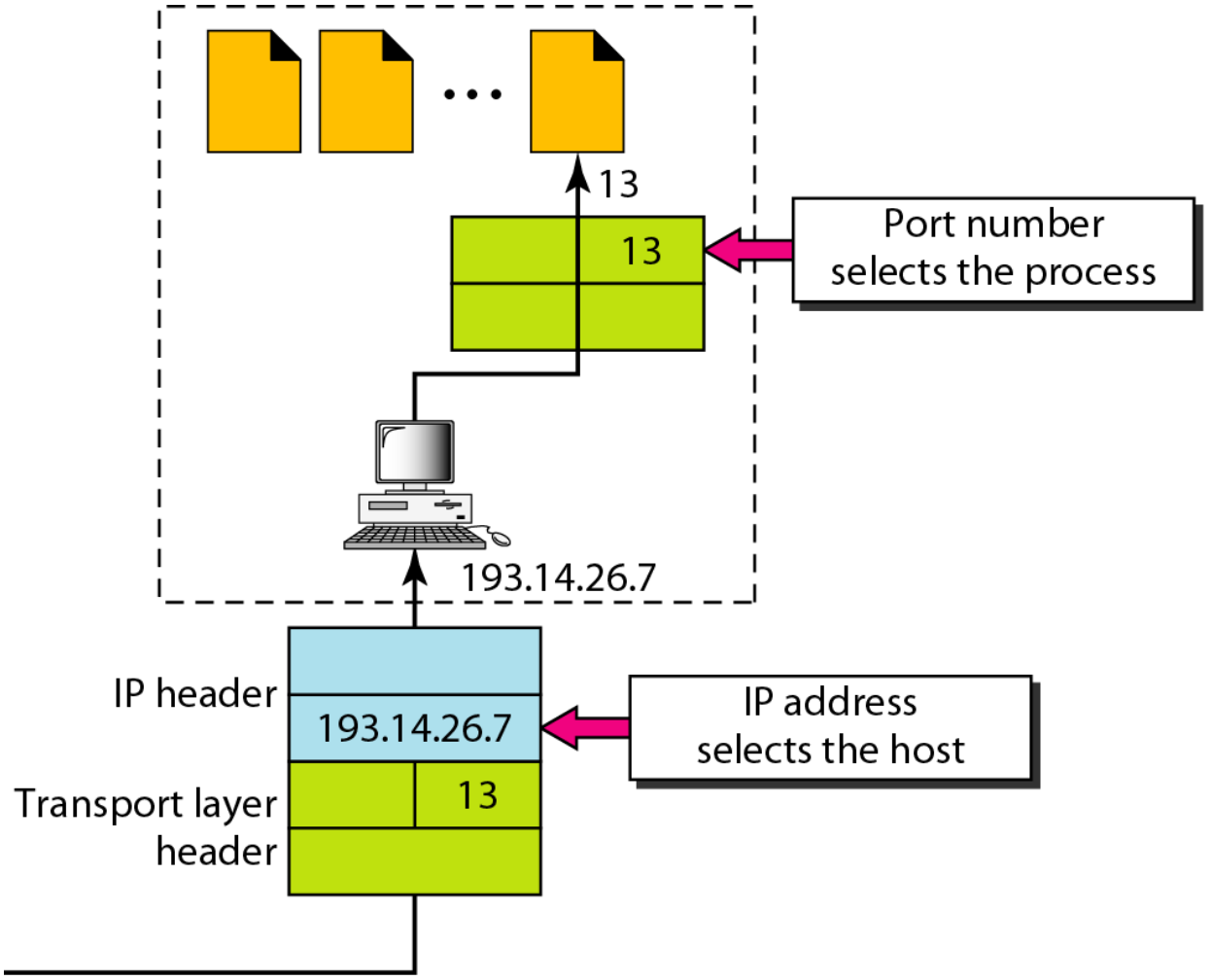
Types of data deliveries



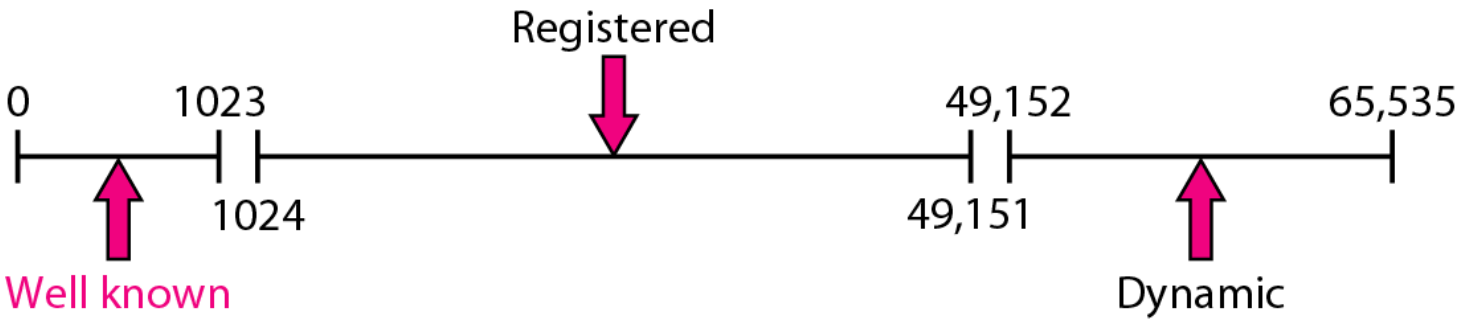
Port numbers



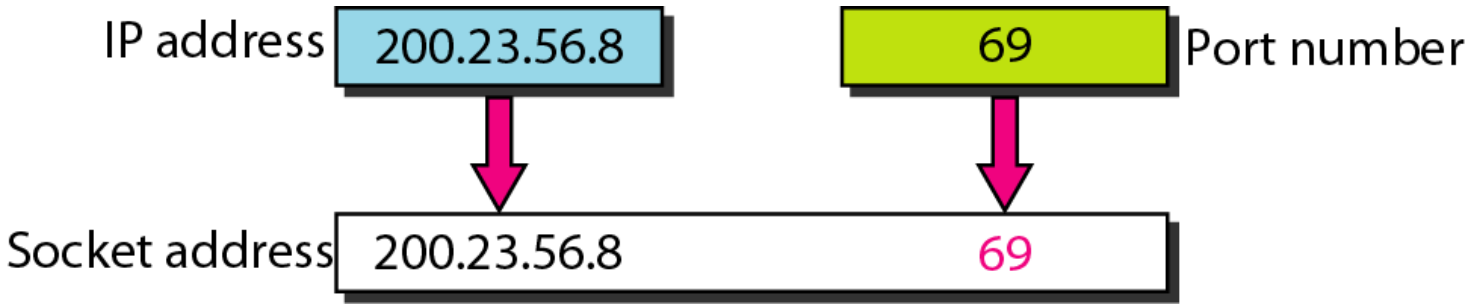
IP addresses versus port numbers



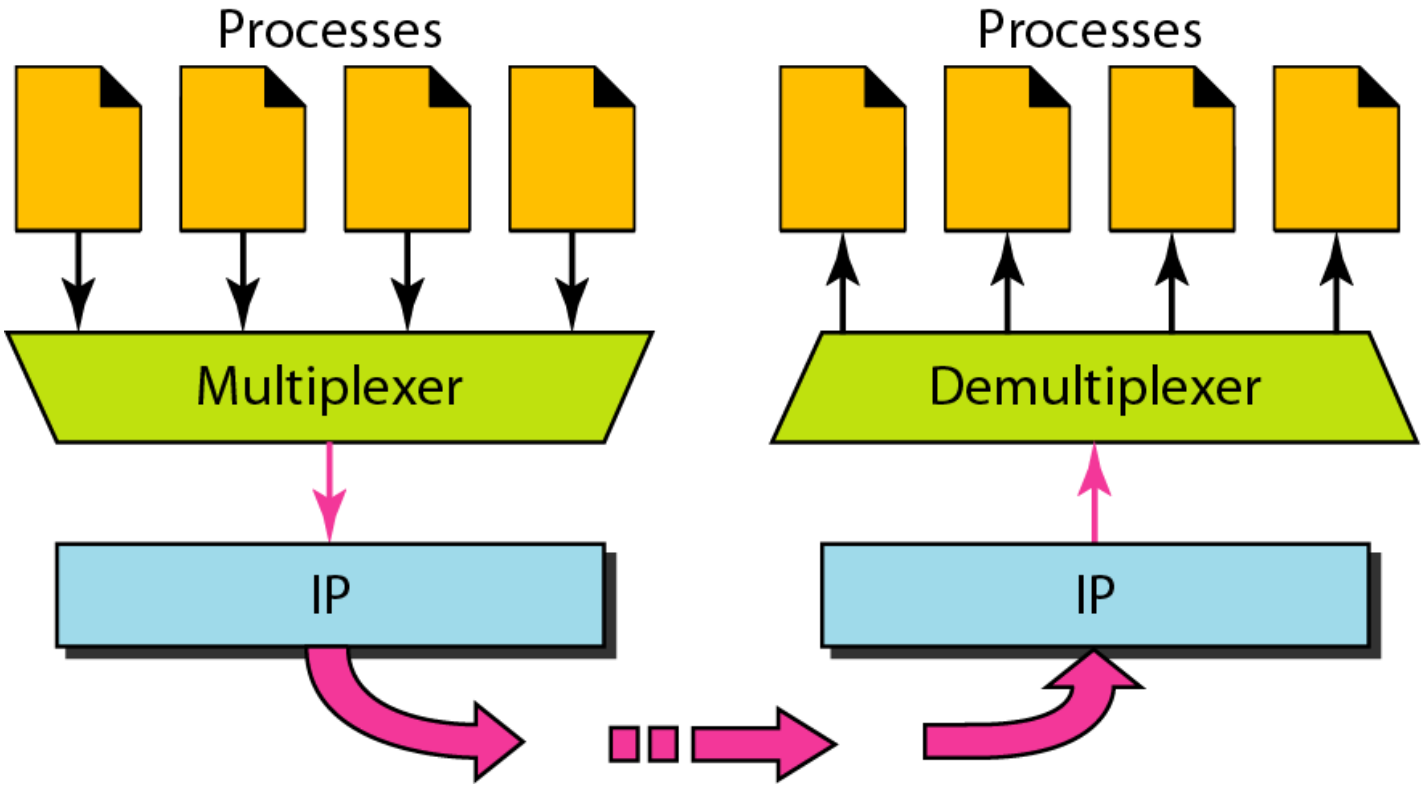
IANA ranges


Registered
0
1023
49,152
65,535
1024
49,151
Well known
Dynamic

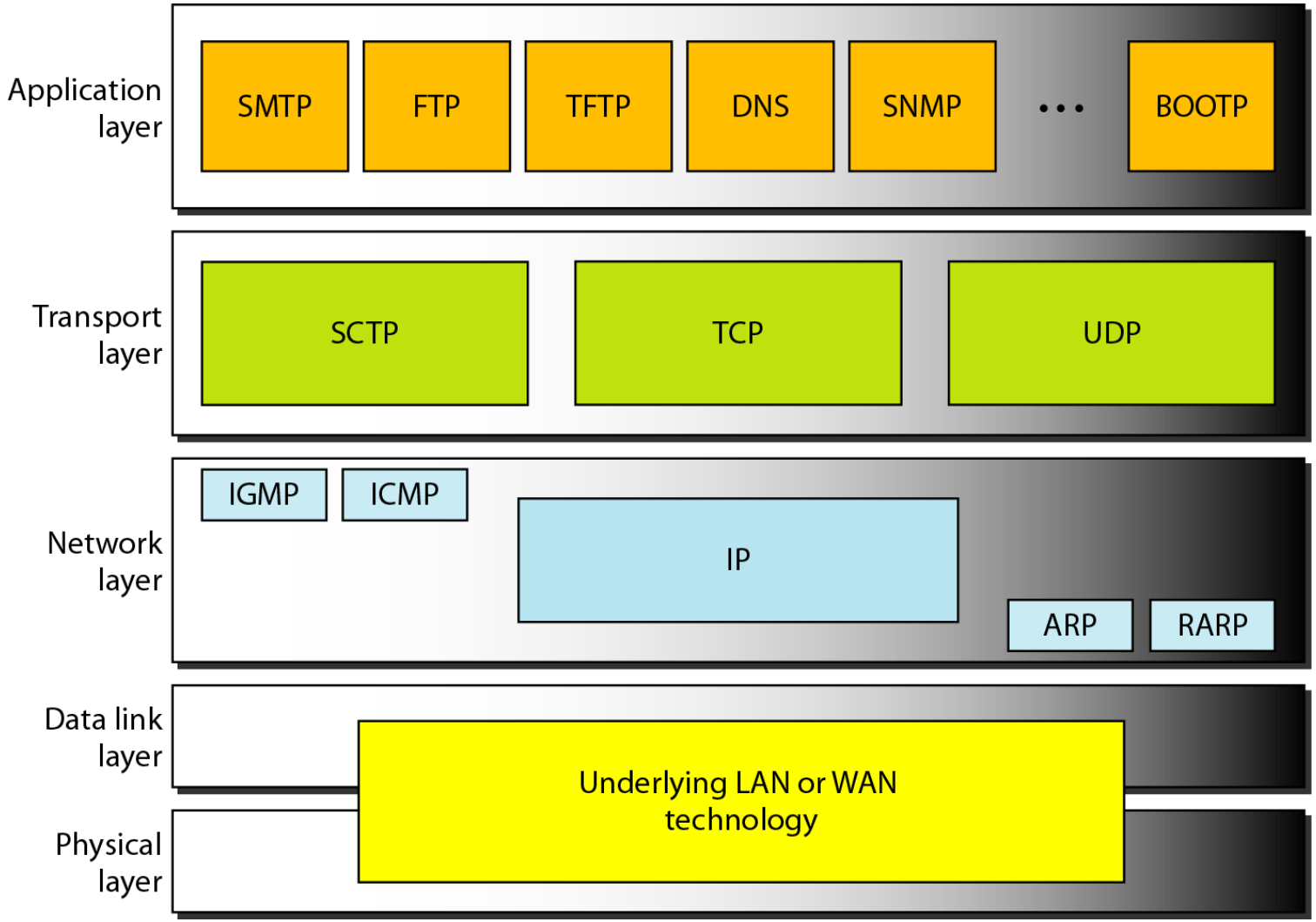
Socket address



Multiplexing and demultiplexing



Position of UDP, TCP, and SCTP in TCP/IP suite



USER DATAGRAM PROTOCOL (UDP)

- *The User Datagram Protocol (UDP) is called a connectionless, unreliable transport protocol*
- *It does not add anything to the services of IP except to provide process-to-process communication instead of host-to-host communication*

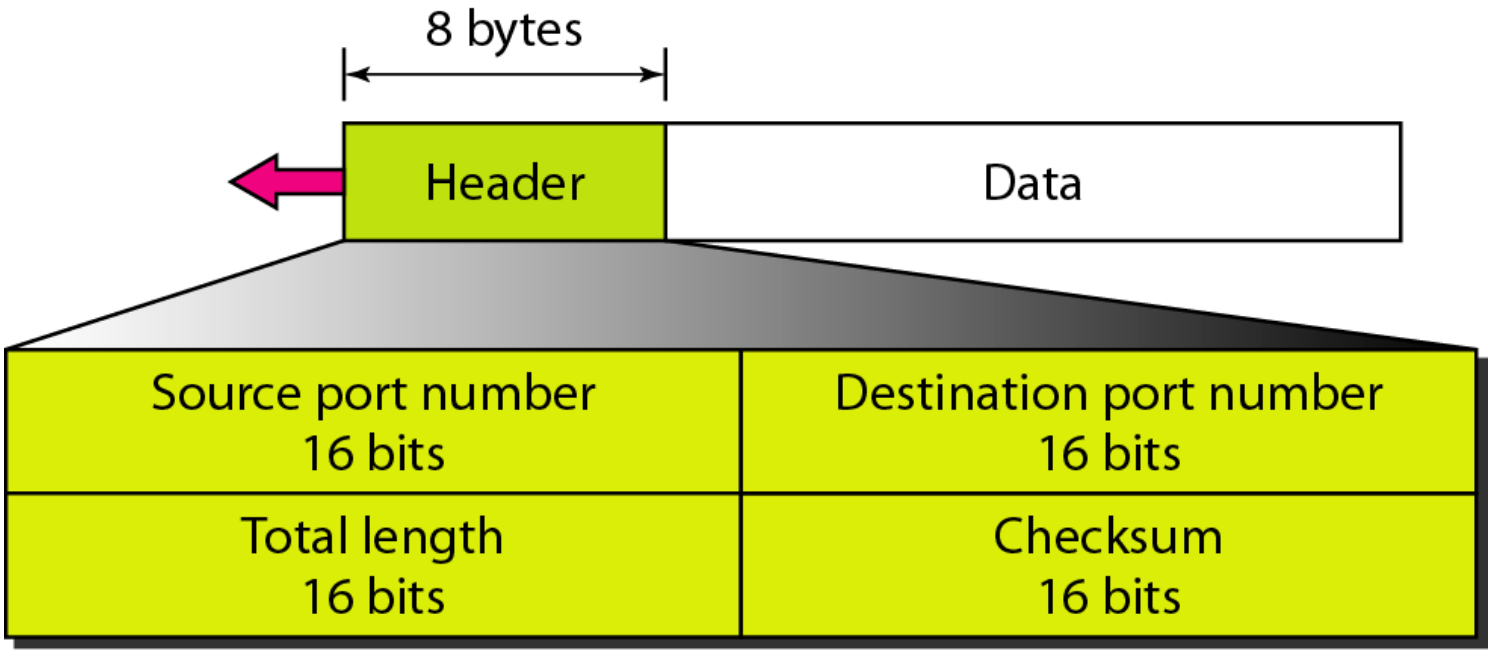
Well-known ports used with UDP

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
53	Nameserver	Domain Name Service
67	BOOTPs	Server port to download bootstrap information
68	BOOTPc	Client port to download bootstrap information
69	TFTP	Trivial File Transfer Protocol
111	RPC	Remote Procedure Call
123	NTP	Network Time Protocol
161	SNMP	Simple Network Management Protocol
162	SNMP	Simple Network Management Protocol (trap)

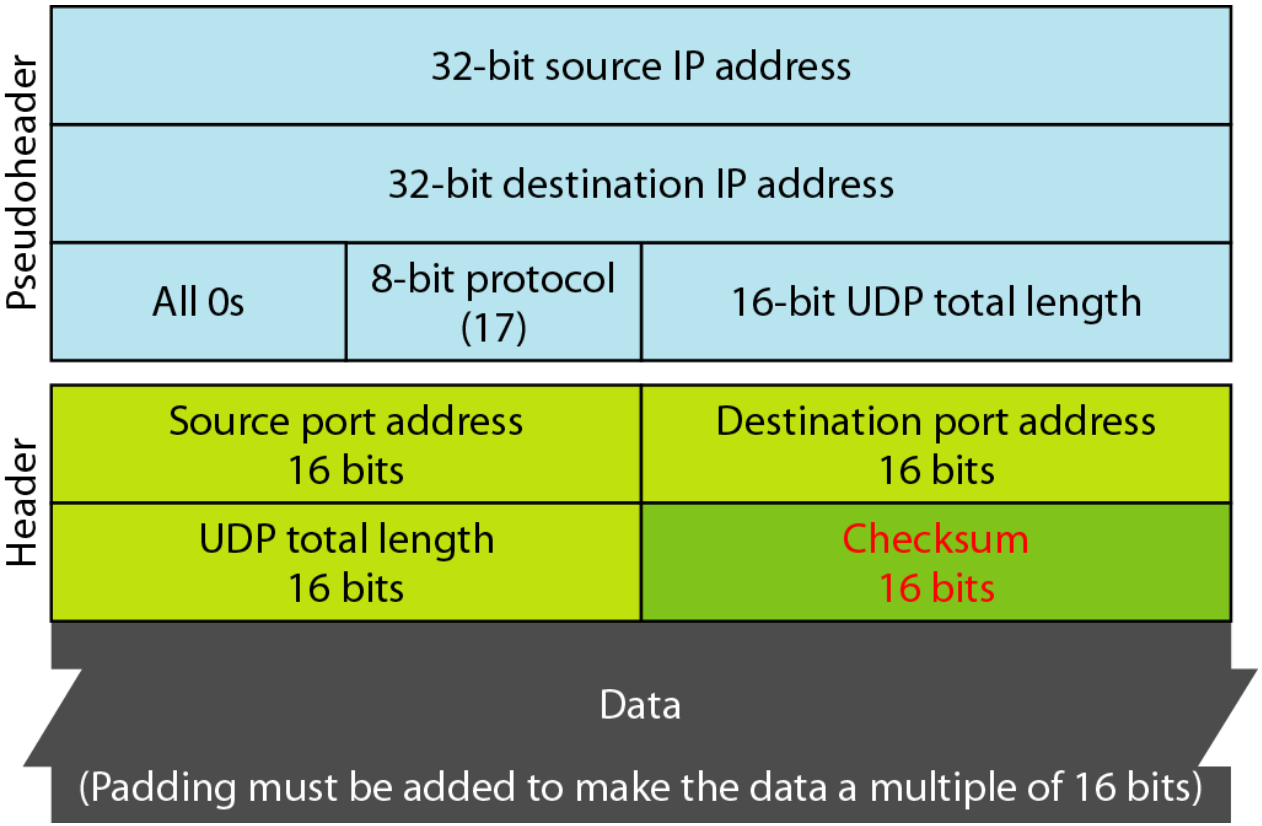
- *In UNIX, the well-known ports are stored in a file called /etc/services.*
- *Each line in this file gives the name of the server and the well-known port number.*
- *We can use the grep utility to extract the line corresponding to the desired application.*
- *The following shows the port for FTP.*
- *Note that FTP can use port 21 with either UDP or TCP.*

```
$ grep    ftp   /etc/services
ftp          21/tcp
ftp          21/udp
```

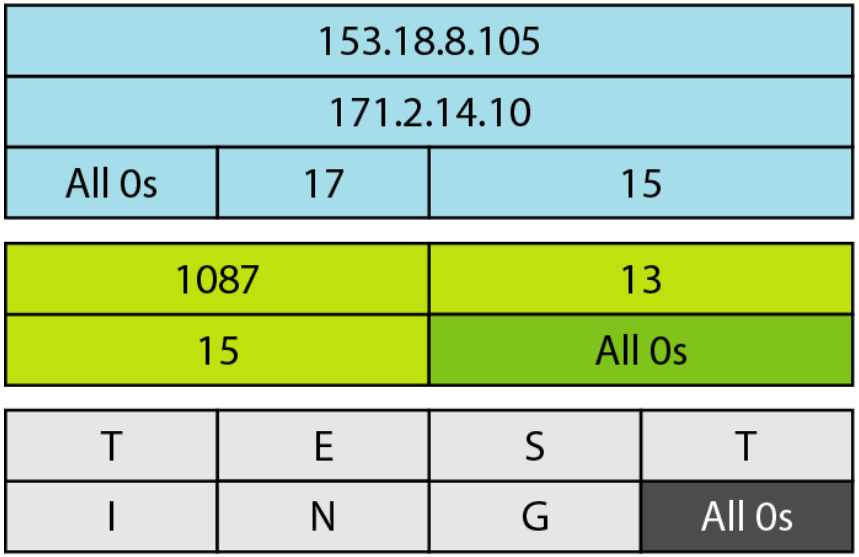
User datagram format



Pseudoheader for checksum calculation

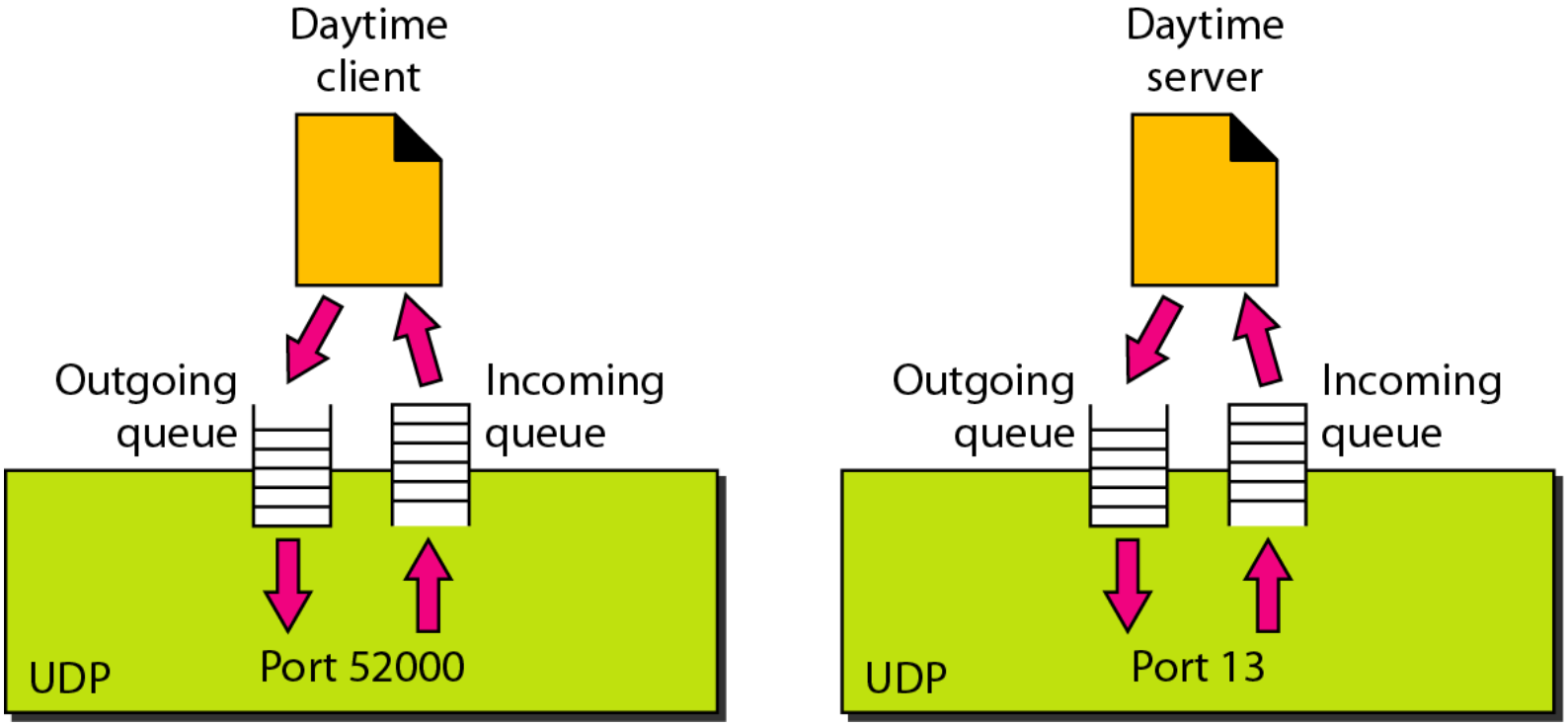


Checksum calculation of a simple UDP user datagram



10011001 00010010	153.18
00001000 01101001	8.105
10101011 00000010	171.2
00001110 00001010	14.10
00000000 00010001	0 and 17
00000000 00001111	15
00000100 00111111	1087
00000000 00001101	13
00000000 00001111	15
00000000 00000000	0 (checksum)
01010100 01000101	T and E
01010011 01010100	S and T
01001001 01001110	I and N
01000111 00000000	G and 0 (padding)
10010110 11101011	Sum
01101001 00010100	Checksum

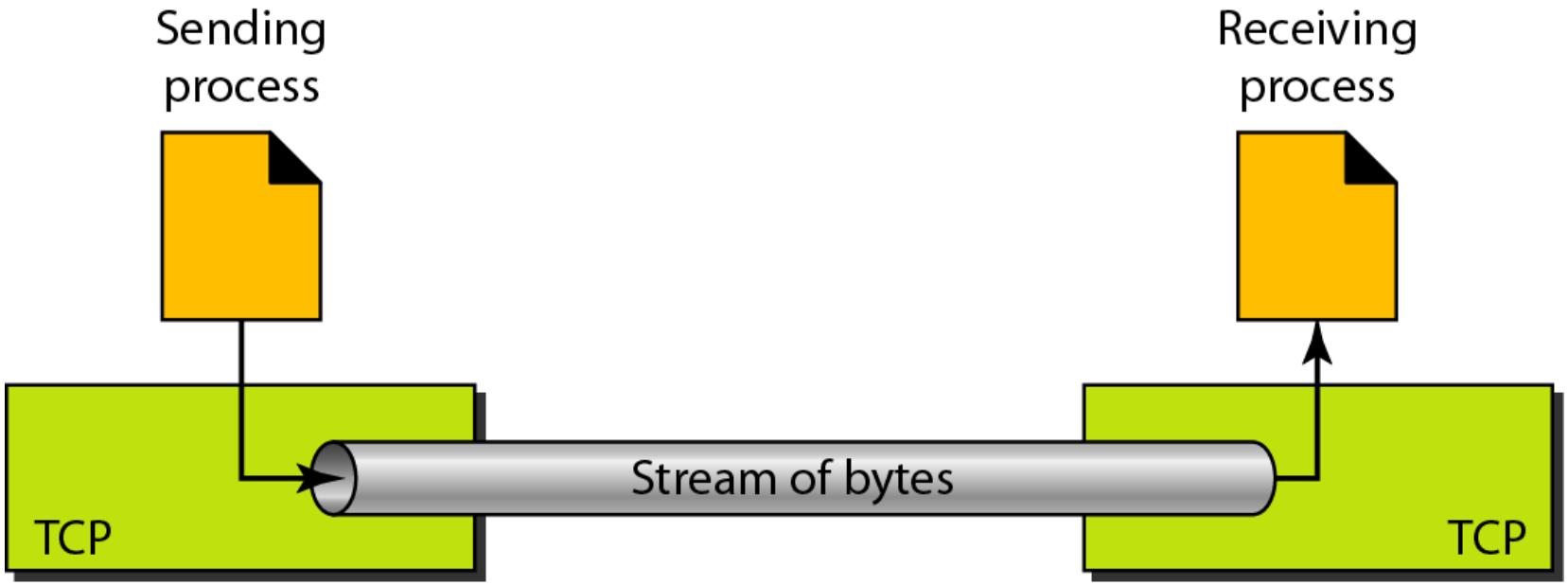
Queues in UDP



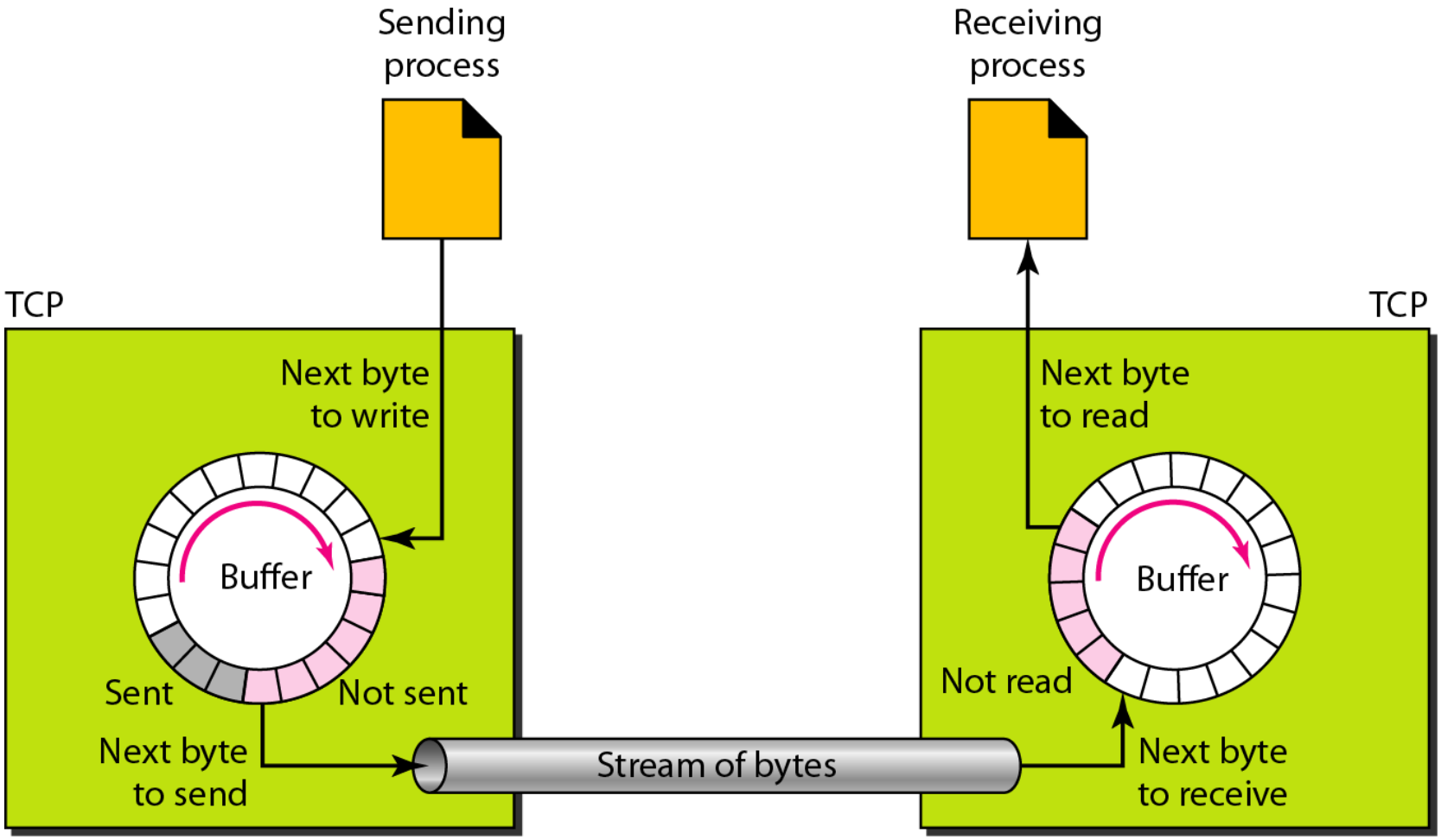
TCP

- *TCP is a connection-oriented protocol*
- *It creates a virtual connection between two TCPs to send data*
- *In addition, TCP uses flow and error control mechanisms at the transport level*

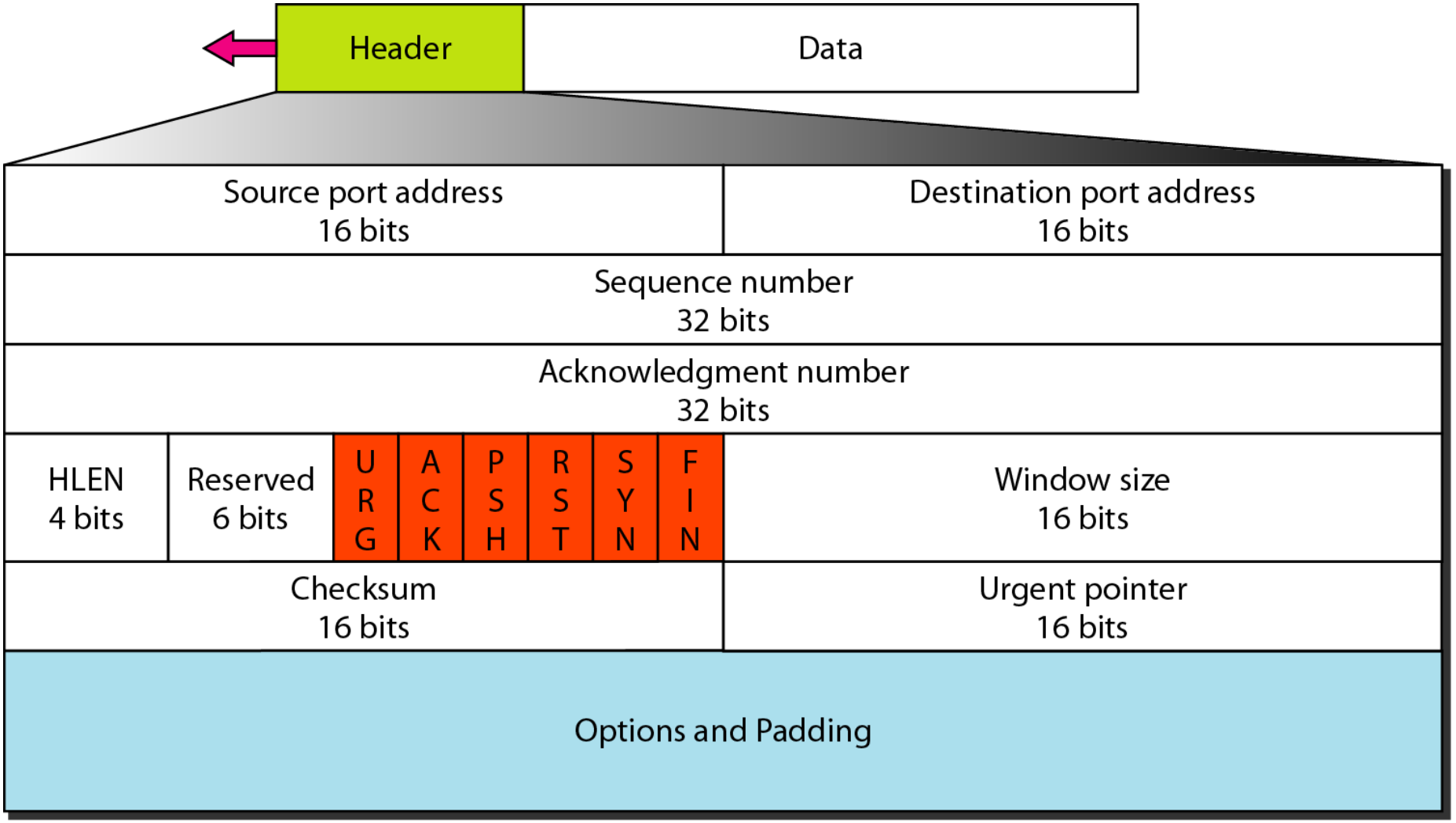
Stream delivery



Sending and receiving buffers



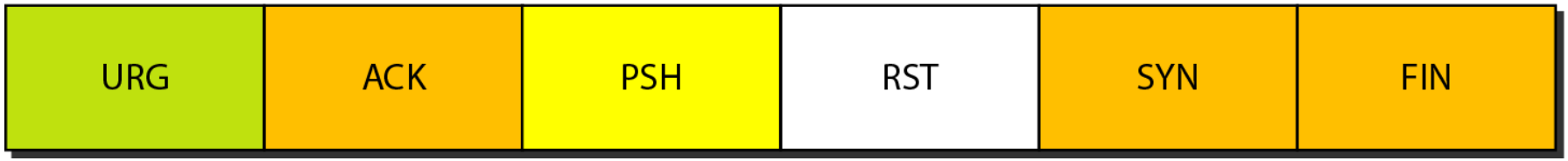
TCP segment format



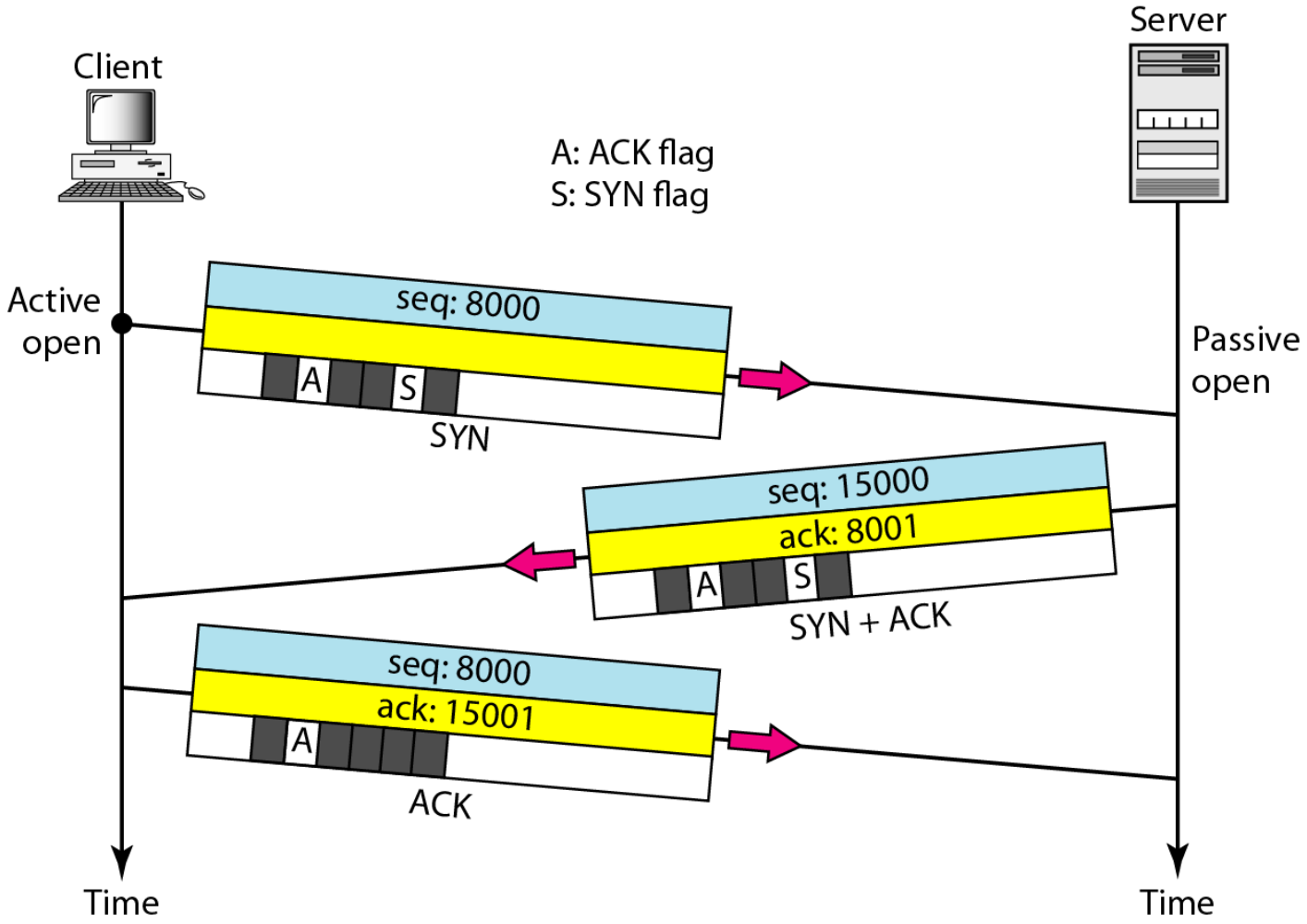
Control field

URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH: Request for push

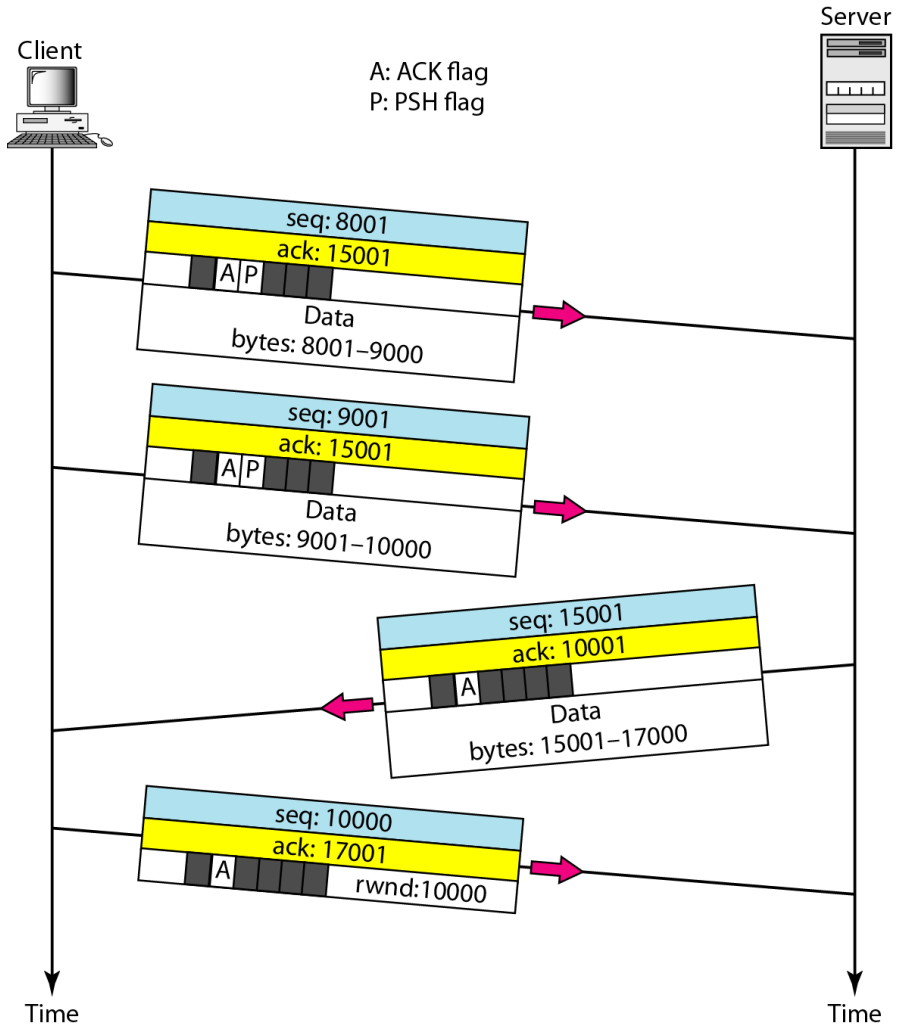
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: Terminate the connection



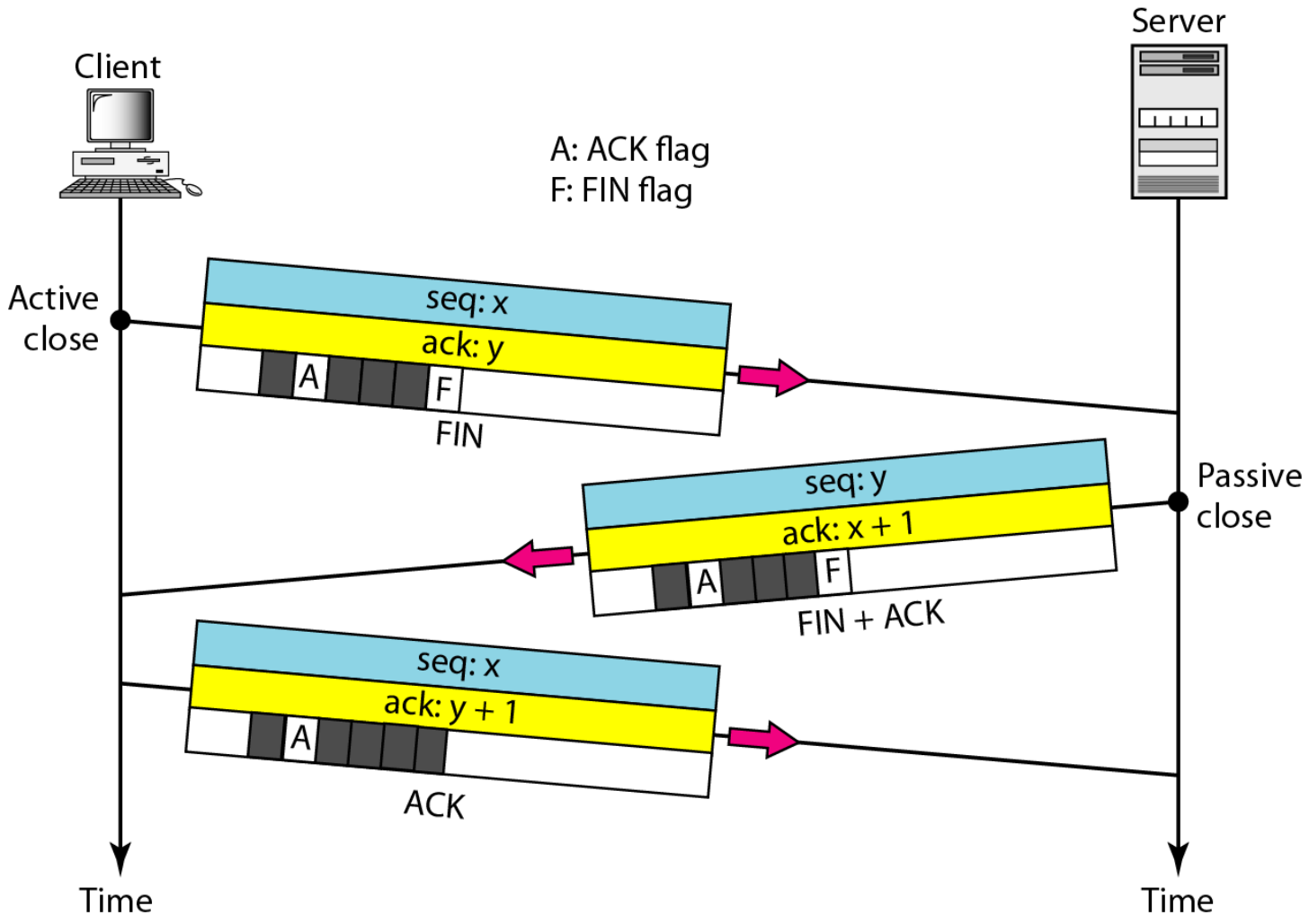
Connection establishment using three-way handshaking



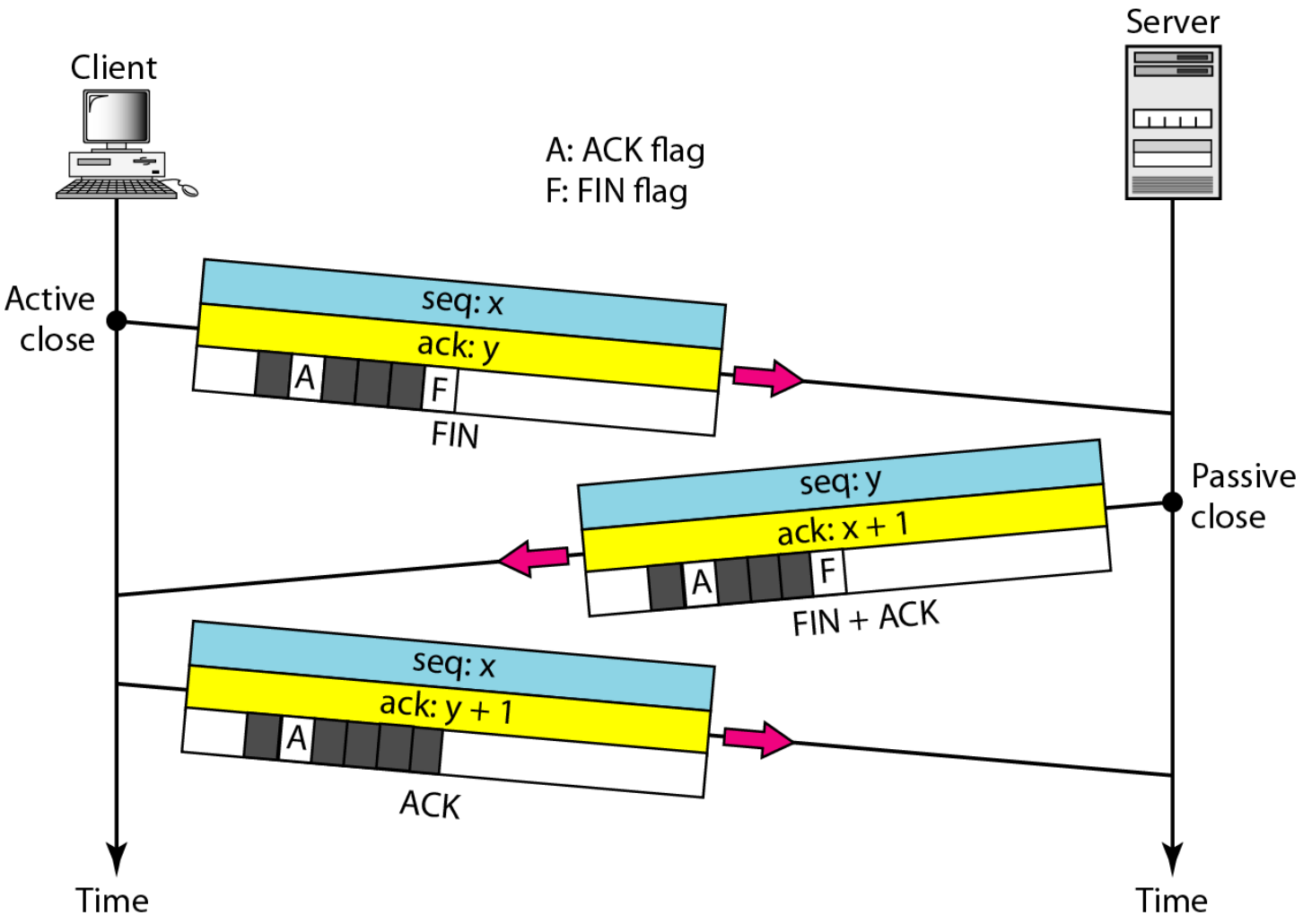
Data transfer



Connection termination using three-way handshaking



Connection termination using three-way handshaking



Thank you