Layering partitions related communications functions into groups that are manageable
Each layer provides a service to the layer above
Each layer operates according to a protocol
Let’s use examples to show what we mean
1. DNS

- User clicks on http://www.nytimes.com/
- URL contains Internet name of machine (www.nytimes.com), but not Internet address
- Internet needs Internet address to send information to a machine
- Browser software uses Domain Name System (DNS) protocol to send query for Internet address
- DNS system responds with Internet address

2. TCP

- Browser software uses HyperText Transfer Protocol (HTTP) to send request for document
- HTTP server waits for requests by listening to a well-known port number (80 for HTTP)
- HTTP client sends request messages through an “ephemeral port number,” e.g. 1127
- HTTP needs a Transmission Control Protocol (TCP) connection between the HTTP client and the HTTP server to transfer messages reliably
3. HTTP

- HTTP client sends its request message: “GET …”
- HTTP server sends a status response: “200 OK”
- HTTP server sends requested file
- Browser displays document
- Clicking a link sets off a chain of events across the Internet!
- Let’s see how protocols & layers come into play…

Protocols

- A protocol is a set of rules that governs
  - how two or more communicating entities in a layer are to interact
  - Messages that can be sent and received
  - Actions that are to be taken when a certain event occurs, e.g. sending or receiving messages, expiry of timers

The purpose of a protocol is to provide a service to the layer above
Layers

- A set of related communication functions that can be managed and grouped together
- Application Layer: communications functions that are used by application programs
  - HTTP, DNS, SMTP (email)
- Transport Layer: end-to-end communications between two processes in two machines
  - TCP, User Datagram Protocol (UDP)
- Network Layer: node-to-node communications between two machines
  - Internet Protocol (IP)

Example: HTTP

- HTTP is an application layer protocol
- Retrieves documents on behalf of a browser application program
- HTTP specifies fields in request messages and response messages
  - Request types; Response codes
  - Content type, options, cookies, …
- HTTP specifies actions to be taken upon receipt of certain messages
Example: TCP

- TCP is a transport layer protocol
- Provides reliable byte stream service between two processes in two computers across the Internet
- Sequence numbers keep track of the bytes that have been transmitted and received
- Error detection and retransmission used to recover from transmission errors and losses
- TCP is connection-oriented: the sender and receiver must first establish an association and set initial sequence numbers before data is transferred
- Connection ID is specified uniquely by 
  (send port #, send IP address, receive port #, receiver IP address)

HTTP uses service of TCP
Example: DNS Protocol

- DNS protocol is an application layer protocol
- DNS is a distributed database that resides in multiple machines in the Internet
- DNS protocol allows queries of different types
  - Name-to-address or Address-to-name
  - Mail exchange
- DNS usually involves short messages and so uses service provided by UDP
- Well-known port 53

- Local Name Server: resolve frequently-used names
  - University department, ISP
  - Contacts Root Name server if it cannot resolve query
- Root Name Servers: 13 globally
  - Resolves query or refers query to Authoritative Name Server
- Authoritative Name Server: last resort
  - Every machine must register its address with at least two authoritative name servers
Example: UDP

- UDP is a transport layer protocol
- Provides *best-effort datagram service* between two processes in two computers across the Internet
- Port numbers distinguish various processes in the same machine
- UDP is *connectionless*
- Datagram is sent immediately
- Quick, simple, but not reliable

How you compare UDP with TCP regarding to advantages and cost?
What kind of applications prefer TCP, or UDP? How about streaming?

Summary

- Layers: related communications functions
  - Application Layer: HTTP, DNS
  - Transport Layer: TCP, UDP
  - Network Layer: IP
- Services: a protocol provides a communications service to the layer above
  - TCP provides connection-oriented reliable byte transfer service
  - UDP provides best-effort datagram service
- Each layer builds on services of lower layers
  - HTTP builds on top of TCP
  - DNS builds on top of UDP
  - TCP and UDP build on top of IP
Why Layering Architectures?

- Layering simplifies design, implementation, and testing by partitioning overall communications process into parts
- Protocol in each layer can be designed separately from those in other layers
- Protocol makes “calls” for services from layer below
- Layering provides flexibility for modifying and evolving protocols and services without having to change layers below
- Monolithic non-layered architectures are costly, inflexible, and soon obsolete

7-Layer OSI Reference Model

- Application Layer
- Presentation Layer
- Session Layer
- Transport Layer
- Network Layer
- Data Link Layer
- Physical Layer

End-to-End Protocols

One or More Network Nodes
**Physical Layer**

- Transfers bits across link
- Definition & specification of the physical aspects of a communications link
  - Mechanical: cable, plugs, pins...
  - Electrical/optical: modulation, signal strength, voltage levels, bit times, …
  - functional/procedural: how to activate, maintain, and deactivate physical links…
- Ethernet, DSL, cable modem, telephone modems…
- Twisted-pair cable, coaxial cable, optical fiber, radio, …

---

**Data Link Layer**

- Transfers *frames* across *direct* connections
  - Groups bits into frames
  - Detection of bit errors; Retransmission of frames
- Activation, maintenance, & deactivation of data link connections
- Medium access control for local area networks
- *Node-to-node* flow control

![Diagram of data link layer](image)
Network Layer

- Transfers *packets* across multiple links and/or multiple networks
  - *Addressing* must scale to large networks
  - Nodes jointly execute *routing* algorithm to determine paths across the network
  - *Forwarding* transfers packet across a node
  - *Congestion control* to deal with traffic surges
  - *Connection* setup, maintenance, and teardown when connection-based

Internetworking

- Internetworking is part of network layer and provides transfer of packets across multiple possibly dissimilar networks
  - Gateways (routers) direct packets across networks

G = gateway
H = host
Transport Layer

- Transfers data end-to-end from process in a machine to process in another machine
  - *Reliable* stream transfer or quick-and-simple single-block transfer
  - Port numbers for addressing (and multiplexing)
  - Message segmentation and reassembly
  - Connection setup, maintenance, and release
  - End-to-end congestion control vs. node-to-node flow control

What data link layer and transport layer have in common and differ?

Application & Upper Layers

- Application Layer: Provides services that are frequently required by applications: DNS, web access, file transfer, email…
- Presentation Layer: Machine-independent representation of data…
- Session Layer: dialog management, recovery from errors, …

\[\text{Incorporated into Application Layer}\]
Headers & Trailers

- Each protocol uses a header that carries addresses, sequence numbers, flag bits, length indicators, etc…

OSI Unified View: Protocols

- Layer $n$ in one machine interacts with layer $n$ in another machine to provide a service to layer $n + 1$
- The entities comprising the corresponding layers on different machines are called peer processes.
- The machines use a set of rules and conventions called the layer-$n$ protocol.
- Layer-$n$ peer processes communicate by exchanging Protocol Data Units (PDUs)
OSI Unified View: Services

- Communication between peer processes is virtual and actually indirect
- Layer \( n+1 \) transfers information by invoking the services provided by layer \( n \)
- Services are available at Service Access Points (SAP’s)
- Each layer passes data & control information to the layer below it until the physical layer is reached and transfer occurs
- The data passed to the layer below is called a Service Data Unit (SDU)
- SDU’s are encapsulated in PDU’s
Connectionless & Connection-Oriented Services

- **Connection-Oriented**
  - Three-phases:
    1. Connection setup between two SAPs to initialize state information
    2. SDU transfer
    3. Connection release
  - E.g. TCP, ATM

- **Connectionless**
  - Immediate SDU transfer
  - No connection setup
  - E.g. UDP, IP

Segmentation & Reassembly

- A layer may impose a limit on the size of a data block that it can transfer for implementation or other reasons
- Thus a layer-\(n\) SDU may be too large to be handled as a single unit by layer-\((n-1)\)
- Sender side: SDU is segmented into multiple PDUs
- Receiver side: SDU is reassembled from sequence of PDUs

What is segmentation & reassembly overhead?
Summary

- Layers: related communications functions
  - Application Layer: HTTP, DNS
  - Transport Layer: TCP, UDP
  - Network Layer: IP

- Services: a protocol provides a communications service to the layer above
  - TCP provides connection-oriented reliable byte transfer service
  - UDP provides best-effort datagram service

- Each layer builds on services of lower layers
  - HTTP builds on top of TCP
  - DNS builds on top of UDP
  - TCP and UDP build on top of IP
Why Internetworking?

- To build a “network of networks” or Internet
  - operating over multiple, coexisting, different network technologies
  - providing ubiquitous connectivity through IP packet transfer
  - achieving huge economies of scale
  - To provide universal communication services

![Diagram of internetworking]

Internet Protocol Approach

- IP packets transfer information across Internet
  - Host A IP → router → router… → router → Host B IP
- IP layer in each router determines next hop (router)
- Network interfaces transfer IP packets across networks
TCP/IP Protocol Suite

**Reliable stream service**
- HTTP
- SMTP
- RTP

**Best-effort connectionless packet transfer**
- DNS
- (ICMP, ARP)

**User datagram service**
- TCP
- UDP

Diverse network technologies

Internet Names & Addresses

**Internet Names**
- Each host a a unique name
  - Independent of physical location
  - Facilitate memorization by humans
  - Domain Name
  - Organization under single administrative unit
- Host Name
  - Name assigned to host computer
- User Name
  - Name assigned to user

leongarcia@comm.utoronto.ca

**Internet Addresses**
- Each host has globally unique logical 32 bit IP address
- Separate address for each physical connection to a network
- Routing decision is done based on destination IP address
- IP address has two parts:
  - netid and hostid
  - netid unique
  - hostid facilitates routing
- Dotted Decimal Notation:
  - int1.int2.int3.int4
  - 128.100.10.13

How to resolve IP name to IP address Mapping?
Physical Addresses

- LANs (and other networks) assign physical addresses to the physical attachment to the network
- The network uses its own address to transfer packets or frames to the appropriate destination
- IP address needs to be resolved to physical address at each IP network interface
- Example: Ethernet uses 48-bit addresses
  - Each Ethernet network interface card (NIC) has globally unique Medium Access Control (MAC) or physical address
  - First 24 bits identify NIC manufacturer; second 24 bits are serial number
  - 00:90:27:96:68:07 12 hex numbers

Intel

Example Internet

<table>
<thead>
<tr>
<th>netid</th>
<th>hostid</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>server</td>
<td>1</td>
<td>1 s</td>
</tr>
<tr>
<td>workstation</td>
<td>1</td>
<td>2 w</td>
</tr>
<tr>
<td>router</td>
<td>1</td>
<td>3 r</td>
</tr>
<tr>
<td>router</td>
<td>2</td>
<td>1 -</td>
</tr>
<tr>
<td>PC</td>
<td>2</td>
<td>2 -</td>
</tr>
</tbody>
</table>

*PPP does not use addresses
Encapsulation

- Ethernet header contains:
  - source and destination physical addresses
  - network protocol type (e.g. IP)

IP packet from workstation to server

1. IP packet has (1,2) IP address for source and (1,1) IP address for destination
2. IP table at workstation indicates (1,1) connected to same network, so IP packet is encapsulated in Ethernet frame with addresses w and s
3. Ethernet frame is broadcast by workstation NIC and captured by server NIC
4. NIC examines protocol type field and then delivers packet to its IP layer
IP packet from server to PC

1. IP packet has (1,1) and (2,2) as IP source and destination addresses
2. IP table at server indicates packet should be sent to router, so IP packet is encapsulated in Ethernet frame with addresses s and r
3. Ethernet frame is broadcast by server NIC and captured by router NIC
4. NIC examines protocol type field and then delivers packet to its IP layer
5. IP layer examines IP packet destination address and determines IP packet should be routed to (2,2)
6. Router’s table indicates (2,2) is directly connected via PPP link
7. IP packet is encapsulated in PPP frame and delivered to PC
8. PPP at PC examines protocol type field and delivers packet to PC IP layer

How the layers work together

(a) PCServer Router HTTP
(b) Server PC

HTTP uses process-to-process reliable byte stream transfer of TCP connection:
Server socket: (IP Address, 80)
PC socket (IP Address, Eph. #)

TCP uses node-to-node unreliable packet transfer of IP
Server IP address & PC IP address

Internet
Routing Table Example (Ex 2.39)

(a) Suppose that all traffic from network 3 that is destined to H1 is to be routed directly through router R2, and all other traffic from network 3 is to go to network 2. What routing table entries should be present in the network 3 hosts and in R2?

Routing Table Example

(b) Suppose that all traffic from network 3 that is destined to H1 is to be routed directly through router R2, and all other traffic from network 3 is to go to network 2. What routing table entries should be present in the network 3 hosts and in R2?

<table>
<thead>
<tr>
<th>H6</th>
<th>H6</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination</td>
<td>Next hop</td>
<td>Destination</td>
</tr>
<tr>
<td>Default</td>
<td>(3,1)</td>
<td>default</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Encapsulation

TCP Header contains source & destination port numbers

IP Header contains source and destination IP addresses; transport protocol type

Ethernet Header contains source & destination MAC addresses; network protocol type

HTTP Request

TCP header

HTTP Request

IP header TCP header HTTP Request FCS

How the layers work together:
Network Analyzer Example

- User clicks on http://www.nytimes.com/
- Ethereal network analyzer captures all frames observed by its Ethernet NIC
- Sequence of frames and contents of frame can be examined in detail down to individual bytes
Top Pane shows frame/packet sequence

Middle Pane shows encapsulation for a given frame

Bottom Pane shows hex & text

---

**Top pane: frame sequence**

- DNS Query
- TCP Connection Setup
- HTTP Request & Response
Summary

- Encapsulation is key to layering
- IP provides for transfer of packets across diverse networks
- TCP and UDP provide universal communications services across the Internet
- Distributed applications that use TCP and UDP can operate over the entire Internet
- Internet names, IP addresses, port numbers, sockets, connections, physical addresses

Chapter 2
Applications and Layered Architectures

Sockets
Socket API

- Berkeley UNIX Sockets API
  - API (Application Programming Interface): provides a standard set of functions that can be called by applications
  - Abstraction for applications to send & receive data
  - Applications create sockets that “plug into” network
  - Applications write/read to/from sockets
  - Implemented in the kernel
  - Facilitates development of network applications
  - Hides details of underlying protocols & mechanisms
- Also in Windows, Linux, and other OS’s

Communications through Sockets

- Application references a socket through a descriptor
- Socket bound to a port number

Diagram:

- Client
  - Application 1
  - Socket
  - Underlying communication protocols
  - User
  - Description
  - Port number
  - User
  - Kernel

- Server
  - Application 2
  - Socket
  - Underlying communication protocols
  - User
  - Description
  - Port number
  - User
  - Kernel

Communications network
Stream mode of service

- Connection-oriented
  - First, setup connection between two peer application processes
  - Then, reliable bidirectional in-sequence transfer of byte stream (boundaries not preserved in transfer)
  - Multiple write/read between peer processes
  - Finally, connection release
  - Uses TCP

- Connectionless
  - Immediate transfer of one block of information (boundaries preserved)
  - No setup overhead & delay
  - Destination address with each block
  - Send/receive to/from multiple peer processes
  - Best-effort service only
    - Possible out-of-order
    - Possible loss
  - Uses UDP

Client & Server Differences

- Server
  - Specifies well-known port # when creating socket
  - May have multiple IP addresses (net interfaces)
  - Waits passively for client requests

- Client
  - Assigned ephemeral port #
  - Initiates communications with server
  - Needs to know server’s IP address & port #
    - DNS for URL & server well-known port #
  - Server learns client’s address & port #
Socket Calls for Connection-Oriented Mode

Server does Passive Open
- `socket` creates socket to `listen` for connection requests
- Server specifies type: TCP (stream)
- `socket` call returns: non-negative integer `descriptor`; or -1 if unsuccessful

```
Server
socket() → bind() → listen() → accept() → Blocks
read() → write() → close()
```

```
Client
socket() → connect() → write() → read() → close()
```

```
Server
socket() → bind() → listen() → accept() → Blocks
read() → write() → close()
```

```
Server
socket() → bind() → listen() → accept() → Blocks
read() → write() → close()
```

```
Client
socket() → connect() → write() → read() → close()
```

Server does Passive Open
- `bind` assigns local address & port # to socket with specified descriptor
- Can wildcard IP address for multiple net interfaces
- `bind` call returns: 0 (success); or -1 (failure)
- Failure if port # already in use or if reuse option not set

```
Server
socket() → bind() → listen() → accept() → Blocks
read() → write() → close()
```

```
Client
socket() → connect() → write() → read() → close()
```
Server does Passive Open
- **listen** indicates to TCP readiness to receive connection requests for socket with given descriptor
- Parameter specifies max number of requests that may be queued while waiting for server to accept them
- **listen** call returns: 0 (success); or -1 (failure)

**Socket Calls for Connection-Oriented Mode**

**Server**
- `socket()`
- `bind()`
- `listen()`
- `accept()` - Blocks
- `read()`
- `write()`
- `close()`

**Client**
- `socket()`
- `connect()`
- `write()`
- `read()`
- `close()`

---

Server calls **accept** to accept incoming requests
- **accept** blocks if queue is empty

**Socket Calls for Connection-Oriented Mode**

**Server**
- `socket()`
- `bind()`
- `listen()`
- `accept()` - Blocks
- `read()`
- `write()`
- `close()`

**Client**
- `socket()`
- `connect()`
- `write()`
- `read()`
- `close()`
Socket Calls for Connection-Oriented Mode

**Server**
- `socket()` creates socket to connect to server
- `Client` specifies type: TCP (stream)
- `socket()` call returns: non-negative integer descriptor; or -1 if unsuccessful

**Blocks**
- `bind()`
- `listen()`
- `accept()`

**Data**
- `read()`
- `write()`
- `close()`

**Client**
- `socket()`
- `connect()` establishes a connection on the local socket with the specified descriptor to the specified remote address and port #
- `connect` returns 0 if successful; -1 if unsuccessful

**Note:** `connect` initiates TCP three-way handshake
Socket Calls for Connection-Oriented Mode

- `accept` wakes with incoming connection request
- `accept` fills client address & port # into address structure
- `accept` call returns: descriptor of new connection socket (success); or -1 (failure)
- Client & server use new socket for data transfer
- Original socket continues to listen for new requests

Server

- `socket()`
- `bind()`
- `listen()`
- `accept()` blocks
- `read()`
- `write()`
- `close()`

Client

- `socket()`
- `connect()`
- `write()`
- `read()`
- `close()`

Socket Calls for Connection-Oriented Mode

Data Transfer

- Client or server call `write` to transmit data into a connected socket
- `write` specifies: socket descriptor; pointer to a buffer; amount of data; flags to control transmission behavior
- `write` call returns: # bytes transferred (success); or -1 (failure); blocks until all data transferred
Socket Calls for Connection-Oriented Mode

**Data Transfer**
- Client or server call `read` to receive data from a connected socket
- `read` specifies: socket descriptor; pointer to a buffer; amount of data
- `read` call returns: # bytes read (success); or -1 (failure); blocks if no data arrives

**Connection Termination**
- Client or server call `close` when socket is no longer needed
- `close` specifies the socket descriptor
- `close` call returns: 0 (success); or -1 (failure)

Note: `write` and `read` can be called multiple times to transfer byte streams in both directions.

Note: `close` initiates TCP graceful close sequence.
Example: TCP Echo Server

```c
/* Example: TCP Echo Server */

#define SERVER_TCP_PORT 3000
#define BUFFER_SIZE 256

int main(int argc, char **argv)
{
    int n, bytes_to_read;
    int sd, new_sd, client_len, port;
    struct sockaddr_in server, client;

    switch(argc) {
        case 1:
            port = SERVER_TCP_PORT;
            break;
        case 2:
            port = atoi(argv[1]);
            break;
        default:
            fprintf(stderr, "Usage: %s [port]
", argv[0]);
            exit(1);
    }

    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "Can't create a socket
");
        exit(1);
    }

    /* Bind an address to the socket */
    if (bind(sd, (struct sockaddr *)&server, sizeof(server)) == -1) {
        fprintf(stderr, "Can't bind name to socket
");
        exit(1);
    }

    /* queue up to 5 connect requests */
    listen(sd, 5);

    while (1) {
        client_len = sizeof(client);
        if ((new_sd = accept(sd, (struct sockaddr *)&client, &client_len)) == -1) {
            fprintf(stderr, "Can't accept client
");
            exit(1);
        }

        bp = buf;
        bytes_to_read = BUFFER_SIZE;
        while ((n = read(new_sd, bp, bytes_to_read)) > 0) {
            bp += n;
            bytes_to_read -= n;
        }

        /* Create a stream socket */
        if (new_sd == socket(AF_INET, SOCK_STREAM, 0) == -1) {
            fprintf(stderr, "Can't create a socket
");
            exit(1);
        }

        printf("Rec'd: %s
", buf);
        write(new_sd, buf, BUFFER_SIZE);
        printf("Sent: %s
", buf);
        close(new_sd);
    }
}
```

Example: TCP Echo Client

```c
/* Example: TCP Echo Client */

#define SERVER_TCP_PORT 3000
#define BUFFER_SIZE 256

int main(int argc, char **argv)
{
    int n, bytes_to_read;
    int sd, port;
    struct sockaddr_in server;
    char *host, *bp, rbuf[BUFFER_SIZE], sbuf[BUFFER_SIZE];

    switch(argc) {
        case 2:
            host = argv[1];
            port = SERVER_TCP_PORT;
            break;
        case 3:
            host = argv[1];
            port = atoi(argv[2]);
            break;
        default:
            fprintf(stderr, "Usage: %s host [port]
", argv[0]);
            exit(1);
    }

    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "Can't create a socket
");
        exit(1);
    }

    if ((hp = gethostbyname(host)) == NULL) {
        fprintf(stderr, "Can't get server's address
");
        exit(1);
    }

    bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);

    /* Connecting to the server */
    if (connect(sd, (struct sockaddr *)&server, sizeof(server)) == -1) {
        fprintf(stderr, "Can't connect
");
        exit(1);
    }

    printf("Connected: server's address is %s
", hp->h_name);
    printf("Transmit:
");
    gets(sbuf);
    write(sd, sbuf, BUFFER_SIZE);
    printf("Receive:
");
    bp = rbuf;
    bytes_to_read = BUFFER_SIZE;
    while ((n = read(sd, bp, bytes_to_read)) > 0) {
        bp += n;
        bytes_to_read -= n;
    }

    printf("Rec'd: %s
", rbuf);
    close(sd);
    return(0);
}
```
Socket Calls for Connection-Less Mode

Server
- `socket()` creates socket of type UDP (datagram)
- `socket()` call returns: descriptor, or -1 if unsuccessful
- `bind()` assigns local address & port # to socket with specified descriptor; Can wildcard IP address
- `recvfrom()` blocks until server receives data from client
- `sendto()` sends data
- `close()` closes socket

Client
- `socket()` creates socket
- `sendto()` sends data
- `recvfrom()` copies bytes received into a specified location
- `recvfrom()` blocks until data arrives
- `close()` closes socket
Socket Calls for Connection-Less Mode

Server
- socket() creates socket of type UDP (datagram)
- socket() returns: descriptor; or -1 if unsuccessful
  
  bind()
  recvfrom()
  sendto()
  close()

Client
- socket()
- sendto() specifies: socket descriptor; pointer to a buffer; amount of data; flags to control transmission behavior; destination address & port #; length of destination address structure
- sendto() returns: # bytes sent; or -1 if unsuccessful
  
  recvfrom()
  close()
Socket Calls for Connection-Less Mode

- `recvfrom` wakes when data arrives
- `recvfrom` specifies: socket descriptor; pointer to a buffer to put data; max # bytes to put in buffer; control flags; copies: sender address & port #: length of sender address structure
- `recvfrom` returns # bytes received or -1 (failure)

Server
- `socket()`
- `bind()`
- `recvfrom()` blocks until server receives data from client
- `sendto()`
- `close()`

Client
- `socket()`
- `sendto()`
- `recvfrom()`
- `close()`

Note: `recvfrom` returns data from at most one `send`, i.e. from one datagram

Socket Calls for Connection-Less Mode

Socket Close
- Client or server call `close` when socket is no longer needed
- `close` specifies the socket descriptor
- `close` call returns: 0 (success); or -1 (failure)

Server
- `socket()`
- `bind()`
- `recvfrom()` blocks until server receives data from client
- `sendto()`
- `close()`

Client
- `socket()`
- `sendto()`
- `recvfrom()`
- `close()`
Example: UDP Echo Server

```c
/* Echo server using UDP */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_UDP_PORT 5000
#define MAXLEN 4096

/* Bind an address to the socket */
int main(int argc, char **argv)
{
    int sd, client_len, port = SERVER_UDP_PORT;
    char buf[MAXLEN];
    struct sockaddr_in server, client;
    bzero((char *)&server, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(sd, (struct sockaddr *)&server,
              sizeof(server)) == -1) {
        fprintf(stderr, "Can't bind same to socket!");
        exit(1);
    }
    /* Create a datagram socket */
    if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        fprintf(stderr, "Can't create a socket!");
        exit(1);
    }
    bzero((char *)&client, sizeof(client));
    /* Bind an address to the client */
    if (argc == 1)
        port = SERVER_UDP_PORT;
    else {
        port = atoi(argv[1]);
        server.sin_port = htons(port);
    }
    if (bfind(sd, (struct sockaddr *)&client,
              sizeof(client)) == -1) {
        fprintf(stderr, "Can't receive datagram!");
        exit(1);
    }
    while (1) {
        client_len = sizeof(client);
        if ((n = recvfrom(sd, buf, MAXLEN, 0,
                           (struct sockaddr *)&client, &client_len)) < 0) {
            fprintf(stderr, "Can't receive datagram!");
            exit(1);
        }
        if (sendto(sd, buf, n, 0,
                   (struct sockaddr *)&client, client_len) != n) {
            fprintf(stderr, "Can't send datagram!");
            exit(1);
        }
        exit(1);
    }
    return 0;
}
```

Example: UDP Echo Client

```c
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_UDP_PORT         5000
#define MAXLEN                  4096
#define DEFLEN                  64

long delay(struct timeval t1, struct timeval t2)
{
    long d;
    d = (t2.tv_sec - t1.tv_sec) * 1000;
    d += ((t2.tv_usec - t1.tv_usec + 500) / 1000);
    return(d);
}

int main(int argc, char **argv)
{
    int     data_size = DEFLEN, port = SERVER_UDP_PORT;
    int     i, j, sd;
    char    *pname, *host, rbuf[MAXLEN], sbuf[MAXLEN];
    struct  hostent         *hp;
    struct  sockaddr_in     server;
    struct  timeval         start, end;
    unsigned long address;
    port = argc > 0 && (strcmp(*argv, "-p") == 0) ?
          atoi(*++argv) : port;
    if (argc < 1)
        port = SERVER_UDP_PORT;
    if (argc > 0) {
        host = *argv;
        if (argc > 0)
            port = atoi(*++argv);
    }
    if (argc > 0)
        data_size = atoi(argv[1]);
    bzero((char *)&server, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    if ((hp = gethostbyname(host)) == NULL) {
        fprintf(stderr, "Can't get server's IP address!");
        exit(1);
    }
    bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);
    if (data_size > MAXLEN) {
        fprintf(stderr, "Data is too big!");
        exit(1);
    }
    for (i = 0; i < data_size; i++)
        sbuf[i] = 'a' + (i < 26) ? i : i % 26;
    gettimeofday(&start, NULL); /* start delay measurement */
    server_len = sizeof(server);
    if (sendto(sd, sbuf, data_size, 0, (struct sockaddr *)
               &server, server_len) == -1) {
        fprintf(stderr, "sendto error!");
        exit(1);
    }
    if (recvfrom(sd, rbuf, MAXLEN, 0, (struct sockaddr *)
                 &server, &server_len) < 0) {
        fprintf(stderr, "recvfrom error!");
        exit(1);
    }
    gettimeofday(&end, NULL); /* end delay measurement */
    printf("round-trip delay=%ld ms.\n", delay(start, end));
    if (strncmp(sbuf, rbuf, data_size) != 0)
        printf("Data is corrupted\n");
    close(sd);
    return(0);
}
```