A Networking Primer

448430
Spring 2009
3/23/2009
Kyoung Shin Park
Multimedia Engineering
Dankook University

Outline

- Network Latency
- Network Bandwidth
- Network Reliability
- Network Protocol
- BSD Sockets & Internet Protocol
  - TCP/IP
  - UDP/IP
  - IP Multicasting

Network Latency
Network Bandwidth
Network Reliability

Network Latency

- Amount of time required to transfer a bit of data from one point to another
- Delay of transfer
- Reasons for network latency
  - speed of light delays (8.25ms of delay per time zone)
  - delays from the computers, network hardware, operating systems
  - delays from the network itself, routers

Network Bandwidth

- The rate at which the network can deliver data to the destination point (bits per second, bps)
- Rate of transfer
- Available bandwidth determined by wire and hardware

Network Reliability

- Measure of how much data is lost by the network during the journey from source to destination host
- Two categories of data loss
  - *Dropping* – the data does not arrive (i.e., discarded by the network)
  - *Corruption* - content of data packets has been changed
- Reliability can vary widely
- When reliability needed send acknowledgement

Network Protocol

- Describes the set of rules that two applications use to communicate with each other
- Consists of three components:
  - Packet format – understanding what the other endpoint is saying
  - Packet semantics – what the recipient can assume when it receives a packet
  - Error behavior – what to do if (when) something goes wrong

Packet Format

- Describes what each type of packet looks like
- Tells the sender what to put in the packet
- Tells recipient how to parse the inbound packet

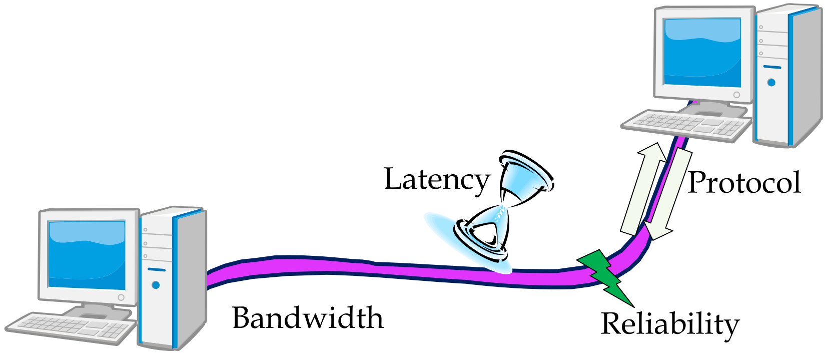
Packet Semantics

- Sender and recipient must agree on what the recipient can assume if it receives a particular packet
- What actions the recipient should take in response to the packet

Error Behavior

- Rules about how each endpoint should respond to various error scenarios

Network Communication



The BSD Sockets Architecture

- *When an application sends a packet, the host must make sure that it gets sent to the right destination, and when a host receives a packet, it must make sure that it is delivered to the correct application. To achieve these two tasks, most hosts on the Internet use the Berkeley Software Distribution (BSD) Sockets network architecture to keep track of applications and network connections.*
- This architecture first gained wide acceptance in the Unix operating system, but today, it is implemented on virtually all of the major commercial operating systems on the market. The WinSock library used on Microsoft Windows 3.1/95/NT platforms is a derivative of the BSD interfaces.

Sockets & Ports

- Socket
  - a software representation of the endpoint to a communication channel
  - can represent many different types of channels (i.e., reliable/unreliable communication, single/multiple destinations, etc)
  - IP address + UDP/TCP + port number
  - 131.120.1.13, UDP, 51
  - 131.120.1.13, TCP, 51
- Port
  - A specific numerical identifier for an individual application

Sockets

- A socket identifies several pieces of information about a communication channel:
  - Protocol: How the operating systems exchange application data
  - Destination host: The destination host address(es) for packets sent on this socket
  - Destination application ID or port: Identifies the appropriate socket on the destination host
  - Source host: Identifies which host is sending the data
  - Local application ID/port: A 16 bit integer that identifies which application is sending data along this socket

Port Numbers

- Provide foundation of open networking
- Like a set of post office box numbers for the protocol
- Each application gets a port number
- Port number + host address gives it a unique identifier to send and receive
- Over 65,000 valid port numbers
  - OS can support many applications at once

Port Numbers

- Port numbers 1 - 1024 are reserved for "well-known" applications/OS services
- 1025 - 10,000 are registered for certain "well-known" protocols
- Example:
  - port 80 is reserved for HTTP
  - port 25 is reserved for simple mail transfer protocol
  - port 1080 is used by SOCKS (network firewall security)

Internet Protocols for Networked VE

- Common Internet Protocols
  - Internet Protocol
  - TCP
  - UDP
- Broadcasting
- Multicasting

Internet Protocols for Networked VE

- Low-level protocol used by hosts and routers to ensure the packets travel from the source to the destination
- Includes facilities for splitting the packets into small fragments
  - network links might not be able to support large packets
  - used to reconstruct packets at other end
- Also includes time to live (TTL) field
  - how many network hops may transfer the packet

Internet Protocols for Networked VE

- Transmission Control Protocol (TCP)
  - Most common protocol in use today
  - Layered on top of IP referred to as TCP/IP
  - Provides illusion of point to point connection to an application running on another machine
  - Each endpoint can regard a TCP/IP connection as a bi-directional stream of bytes between two endpoints
  - Application can detect when other end of connection has gone away/disconnected

User Datagram Protocol (UDP)

- The User Datagram Protocol (UDP) is a lightweight communication protocol
- Differs from TCP in three respects:
  - connection-less transmission
  - best-efforts delivery
  - packet-based data semantics
- Does not establish peer-to-peer connections

User Datagram Protocol (UDP)

- Sender and recipient of do not keep any information about the state of the communication session between the two hosts
- Simply provides **best-efforts delivery**, i.e. no guarantee that data is delivered reliably or in order
- Endpoints do not maintain state information about the communication, UDP data is sent and received on a **packet-by-packet basis**
- **Datagrams** must not be too big, because if they must be fragmented, some pieces might get lost in transit

User Datagram Protocol (UDP) Advantages

- Simplicity
- Does not include the overhead needed to detect reliability and maintain connection-oriented semantics
  - UDP packets require considerably less processing at the transmitting and receiving hosts
- Does not maintain the illusion of a data stream
  - packets can be transmitted as soon as they are sent by the application instead of waiting in line behind other data in the stream; similarly, data can be delivered to the application as soon as it arrives at the receiving host instead of waiting in line behind missing data

User Datagram Protocol (UDP) Advantages

- Many operating systems impose limits on how many simultaneous TCP/IP connections they can support.
- Operating system does not need to keep UDP connection information for every peer host, UDP/IP is more appropriate for large-scale distributed systems where each host communicates with many destinations simultaneously

UDP Disadvantage for Some NVEs

- When a socket is receiving data on a UDP port, it will receive packets sent to it by any host, whether it is participating in the application or not
- This possibility can represent a security problem for some applications that do not robustly distinguish between expected and unexpected packets
- For this reason, many network firewall administrators block UDP data from being sent to a protected host from outside the security perimeter

UDP Broadcasting

- With UDP/IP, an application can direct a packet to be sent to one other application endpoint
- Could send the same packet to multiple destinations by repeatedly calling **sendto()** (in C) or **DatagramSocket.send()** (in Java)
- This approach has two disadvantages:
  - Excessive network bandwidth is required because the same packet is sent over the network multiple times
  - Each host must maintain an up-to-date list of all other application endpoints who are interested in its data

UDP Broadcasting

- UDP broadcasting provides a partial solution to these issues
- Allows a single transmission to be delivered to all applications on a network who are receiving on a particular port
- Useful for small net-VE's
- Expensive because every host on network must receive and process every broadcast packet
- Not used for large or internet based VE's (use IP Multicast)

IP Multicasting

- UDP broadcasting can only be used in a LAN environment
- Even if no application on that host is actually interested in receiving the packet each host on the LAN must:
  - receive packet
  - process the packet
- Multicasting is the solution to both of these concerns
- Appropriate for Internet use, as well as LAN use
- Does not impose burdens on hosts that are not interested in receiving the multicast data

IP Multicasting

- IP addresses in the range 224.0.0.0 through 239.255.255.255 are designated as multicast addresses
- The 224.*.*.* addresses are reserved for use by the management protocols on a LAN, and packets sent to the 239.*.*.* addresses are typically only sent to hosts within a single organization
- Internet-based net-VE application should therefore use one or more random addresses in the 225.*.*.* to 238.*.*.* range
- The sender transmits data to a multicast IP address, and a subscriber receives the packet if it has explicitly joined that address

IP Multicasting

- Rapidly emerging as the recommended way to build large-scale net-VEs over the Internet
- Provides:
  - desirable network efficiency
  - allows the net-VE to partition different types of data by using multiple multicast addresses
- Using a well-known multicast address, net-VE participants can announce their presence and learn about the presence of other participants

IP Multicasting

- Also an appropriate technique for discovering the availability of other NVE resources such as terrain servers
- These features make multicasting desirable even for LAN-based NVEs.

IP Multicasting Limitations

- Limitations generally related to its infancy
- Although an increasing number of routers are multicast-capable, many older routers are still not capable of handling multicast subscriptions
- In the meantime, multicast-aware routers communicate directly with each other, "tunneling" data past the routers that cannot handle multicast data

Selecting an NVE Protocol

- Multiple protocols can be used in a single system
- Not which protocol should I use in my NVE but which protocol should I use to transmit this piece of information?
- Using TCP/IP
  - Reliable data transmission between two hosts
  - Packets are delivered in order, error handling
  - Relatively easy to use
  - Point-to-point limits its use in large-scale NVEs
  - Bandwidth overhead

Selecting an NVE Protocol

- Using UDP/IP
  - Lightweight
  - Offers no reliability nor guarantees the order of packets
  - Packets can be sent to multiple hosts
  - Deliver time-sensitive information among a large number of hosts
  - More complex services have to be implemented in the application
    - Serial numbers, timestamps
  - Recovery of lost packets
    - Positive acknowledgement scheme
    - Negative acknowledgement scheme
    - More effective when the destination knows the sources and their frequency
  - Transmit a quench packet if packets are received too often

Selecting an NVE Protocol

- Using IP Broadcasting
  - Design considerations similar to UNICAST UDP/IP
  - Limited to LAN
  - Not for NVEs with a large number of participants
  - To distinguish different applications using the same port number (or multicast address)
    - Avoid the problem entirely – assign the necessary number
    - Detect conflict and renegotate – notify the participants and direct them to migrate a new port number
    - Use protocol and instance magic numbers – each packet includes a magic number at a well-known position
    - Use encryption

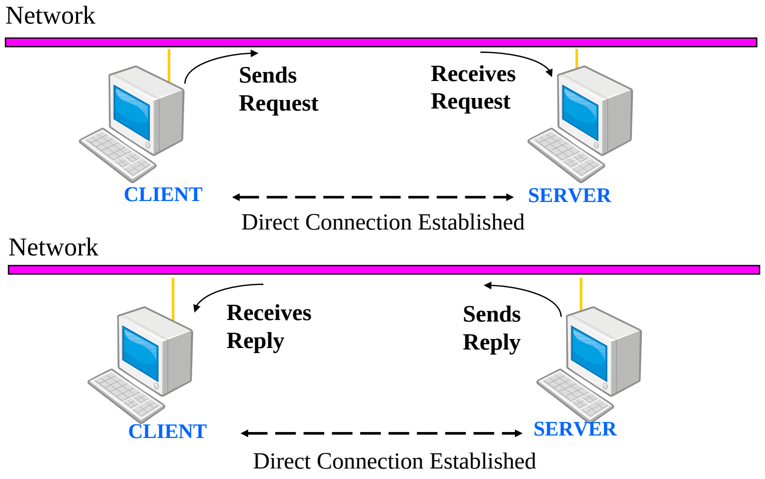
Selecting an NVE Protocol

- Using IP Multicasting
  - Provides a quite efficient way to transmit information among a large number of hosts
  - Information delivery is restricted
    - Time-to-live
    - Group subscription
  - Preferred method for large-scale NVEs
  - How to separate the information flow among different multicast groups
    - A single group/address for all information
    - Several multicast groups to segment the information

CODE

- C/C++ for:
  - TCP/IP
  - UDP/IP
  - BROADCAST
  - MULTICAST

TCP/IP Client-Server Model



Network

Sends Request

Receives Request

CLIENT

SERVER

Direct Connection Established

Network

Receives Reply

Sends Reply

CLIENT

SERVER

Direct Connection Established

C / C++ TCP/IP Socket Implementation

CLIENT ACTIONS:

1. **Obtain a socket**
2. **Connect to the server**
3. **Communicate with server**
   - *** Send data/requests**
   - *** Receive data/replys**
4. **Close the socket**

SERVER ACTIONS:

1. **Obtain a socket**
2. **Bind the socket to a 'well known' port**
3. **Receive connections from clients**
4. **Communicate with clients**
   - *** Receive data/requests**
   - *** Send data/replys**
5. **Close the socket**

C / C++ TCP/IP Socket Implementation

CLIENT ACTIONS:

1. Obtain a socket
2. Connect to the server
3. Communicate with server
   - * Send data/requests
   - * Receive data/replys
4. Close the socket

OBTAIN A SOCKET

- * Use 'socket(...)' function
- * 'socket(...)' interfaces with the O/S to create a socket
- * Arguments in call to socket() determine the protocol and data stream semantics
- * 'socket(...)' returns an int that the user can use to reference the socket

C / C++ TCP/IP Socket Implementation

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int sock;                              //  user reference to the socket

// Allocate a socket function call parameters
//     PF_INET: Use the Internet family of Protocols
//     SOCK_STREAM: Provide reliable byte-stream semantics
//     0: Use the default protocol (TCP) */
sock = socket(PF_INET, SOCK_STREAM, 0);

if (sock == -1) {                      // an error has occured
       perror("socket");
       return;
}
```

C / C++ TCP/IP Socket Implementation

CLIENT ACTIONS:

1. Obtain a socket
2. **Connect to the server**
3. Communicate with server
   - * Send data/requests
   - * Receive data/replys
4. Close the socket

CONNECT TO THE SERVER

* Allocate an Internet Socket Address
- sockaddr_in
- contains server address and port

* Connect to the server
- bind a free local port to the client's socket
- attempt to connect to the server specified in sockaddr_in
- if connection is successful, it is initialized

C / C++ TCP/IP Socket Implementation

```
struct sockaddr_in serverAddr;                    // The address and port of the server

bzero((char *)&serverAddr, sizeof(serverAddr));          // Zero out allocated memory
serverAddr.sin_family = PF_INET;                          // Use Internet addresses
serverAddr.sin_addr.s_addr = inet_addr("10.25.43.9");

//  The inet_addr() function converts  an IP address string into a four-byte
//   integer with one byte for each of the address values
//   htons() converts a 16-bit short integer into the network byte order so
//   that other hosts can interpret the integer even if they internally store
//   integers using a different byte order
serverAddr.sin_port = htons(13214);

//  Connect to the remote host
if (connect(sock, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) == -1) {
            perror("connect");
            return;
}
```

C / C++ TCP/IP Socket Implementation

CLIENT ACTIONS:

1. Obtain a socket
2. Connect to the server
3. **Communicate with server**
   - * Send data/requests
   - * Receive data/replys
4. Close the socket

COMMUNICATE W/ SERVER

* Place data to send into a buffer

* Provide the buffer to the O/S along with socket ID for transmission

C / C++ TCP/IP Socket Implementation

```
int BUFFERLEN = 255;
char buf[BUFFERLEN];                                      // Allocate a buffer

sprintf(buf, "%chello!", (char)strlen("hello!"));   // Write data to buffer

if (write(sock, buf, 1+strlen(buf)) == -1) {          // Write buffer to socket
            perror("write");                              //  i.e. send the data
            return;
}
```

C / C++ TCP/IP Socket Implementation

CLIENT ACTIONS:

1. Obtain a socket
2. Connect to the server
3. Communicate with server
   - * Send data/requests
   - * Receive data/replys
4. Close the socket

CLOSE THE SOCKET

* Invoke ‘close(...)’ on the socket

* Both sides must close their sockets to completely close the connection

* code == > close(sock);

C / C++ TCP/IP Socket Implementation

SERVER ACTIONS:

1. Obtain a socket
2. Bind the socket to a ‘well known’ port
3. Receive connections from clients
4. Communicate with clients
   - * Recieve data/requests
   - * Send data/replys
5. Close the socket

BIND THE SOCKET TO A PORT

* Allocate an Internet Socket Address structure
- sockaddr_in
- contains address and port of the server

* Bind the server to the socket

C / C++ TCP/IP Socket Implementation

```
struct sockaddr_in serverAddr;                    //  The address and port of the server

bzero((char *)&serverAddr, sizeof(serverAddr)); //  Zero out allocated memory
serverAddr.sin_family = PF_INET;                  //  Use Internet addresses

//  INADDR_ANY says that the operating system may choose to which local IP address
//  to attach the application. For most machines, which only have one address, this
//  simply chooses that address. The  htonl()  function converts a four-byte integer long
//  integer into the network byte order so that other hosts can interpret the integer
//  even if they internally store  integers using a different byte order
serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
serverAddr.sin_port = htons(13214);

//  Bind the socket to the well-known port
if (bind(sock, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) == -1) {
      perror("bind");
      return;
}
```

C / C++ TCP/IP Socket Implementation

SERVER ACTIONS:

1. Obtain a socket
2. Bind the socket to a ‘well known’ port
3. Receive connections from clients
4. Communicate with clients
   - * Recieve data/requests
   - * Send data/replys
5. Close the socket

RECEIVE CLIENT CONNECTIONS

* Listen for client connections
- use ‘listen(...)’
- Tell O/S how many client connections can be queued

* Call ‘accept(...)’ to wait for a client to connect

C / C++ TCP/IP Socket Implementation

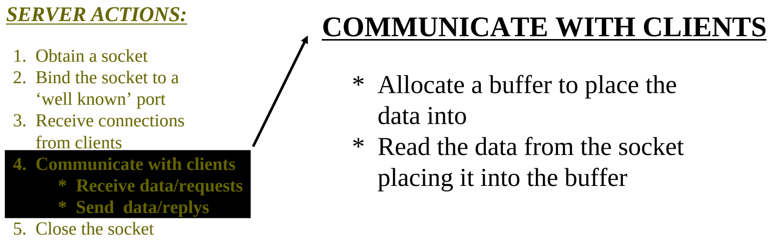
```
int acceptSock = sock;   //  The original socket allocated by the server is used to
                                            //  listen for and accept client connections
struct sockaddr_in clientAddr;  //  allocate an address structure for the clients address

listen(acceptSock, 4);                      //  listen for connections

while ((sock = accept(acceptSock, (struct sockaddr)&clientAddr, sizeof(clientAddr)))
!= -1) {
// sock represents a connection to a client, clientAddr is the client's  host address
// and port
/* ... Process client connection ... */
}

// Only break out of loop if there is an error
perror("accept");
```

C / C++ TCP/IP Socket Implementation



SERVER ACTIONS:

1. Obtain a socket
2. Bind the socket to a 'well known' port
3. Receive connections from clients
4. **Communicate with clients**
   - *** Receive data/requests**
   - *** Send data/replys**
5. Close the socket

COMMUNICATE WITH CLIENTS

* Allocate a buffer to place the data into

* Read the data from the socket placing it into the buffer

C / C++ TCP/IP Socket Implementation

```
int BUFFERLEN = 255;              // Allocate buffer to place received data in
char buf[BUFFERLEN];
int byteCount = 0;                // Total number of bytes read
int n;                            // Number of bytes read this time

while (((n = read(sock, buf+byteCount, BUFFERLEN-byteCount)) > 0) {
      byteCount += n;
      if (byteCount > buf[0]) {
            break;
      }
}
if (n < 0) {                        // error
      perror("read");
      return;
}
if (n == 0) {                       // Connection was closed
      /* ... */
}
```

C / C++ TCP/IP Socket Implementation

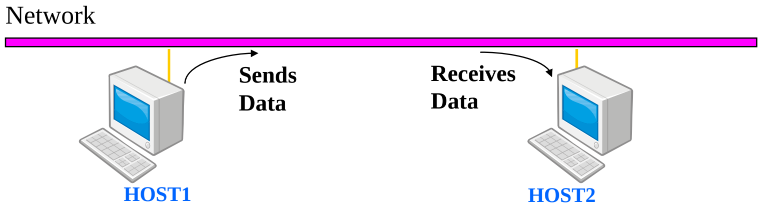
NOTES:

* The server has actually opened two sockets,
  - One to receive connecting clients on
  - One to actually communicate to a specific client on

* Provided code can only process one client at a time
  - threads can be used to process multiple client connections

* read() and accept() calls block until data or a new client connection have arrived
  - This can be avoided using the select() function
  - Code is provided in the book

UDP/IP Communication Model



Network

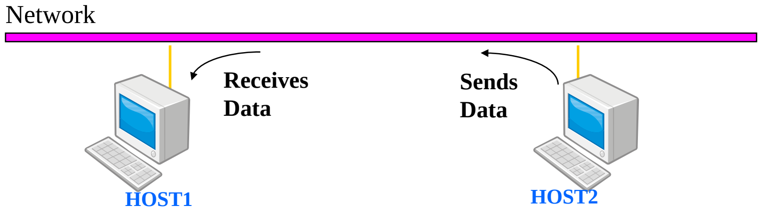
Sends Data

Receives Data

HOST1

HOST2

No Connection Established



Network

Receives Data

Sends Data

HOST1

HOST2

No Connection Established

C / C++ UDP/IP Socket Implementation

STEPS TO IMPLEMENT A UDP/IP SOCKET

1) **Obtain a socket**
2) **Bind the socket to a ‘well known’ port**
3) **Transmit Data**
4) **Receive Data**
5) **Close the socket**

*** Above process is ‘a way’ not the only way**

C / C++ UDP/IP Socket Implementation

STEP 1: OBTAIN A SOCKET

- **Use the ‘socket(...)’ function, ‘socket(...)’ interfaces with the O/S to create a socket**
- **Arguments in the call to ‘socket(...)’ determine the protocol and data stream semantics**
- **Call to ‘socket(...)’ returns an int that user can use to reference the socket**
- **No call to connect() is required as in TCP/IP because UDP/IP is connectionless**

C / C++ UDP/IP Socket Implementation

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int sock; // Declare an int to hold a reference to a socket

//  arguments in call to socket are as follows...
//  PF_INET: Use the Internet family of Protocols
//  SOCK_DGRAM: Provide best-efforts packet semantics
//  0: Use the default protocol (UDP)

//  create/open the socket
sock = socket(PF_INET, SOCK_DGRAM, 0);
if (sock == -1) {
     perror("socket");
     return;
}
```

C / C++ UDP/IP Socket Implementation

STEP 2: BIND SOCKET TO A 'WELL KNOWN' PORT

* **When data is first transmitted through a socket the O/S binds a randomly chosen port to the socket**
* **It is better to bind the socket to a 'well known' port so other hosts know where to send data**
* **Allocate an internet address structure to hold the sender's IP address and port number (sockaddr_in)**
* **Bind the socket to the port contained in the internet address structure**

C / C++ UDP/IP Socket Implementation

```
struct sockaddr_in localAddr;           // Allocate  an internet address structure
                                        // for the  address/port of the local endpoint

bzero((char *)&localAddr, sizeof(localAddr));          // zero out allocated memory
localAddr.sin_family = PF_INET;                        // Use Internet addresses
localAddr.sin_addr.s_addr = htonl(INADDR_ANY); // Use any local IP address
localAddr.sin_port = htons(13214);                     // Port that others can send to

// Bind the socket to the well-known port
if (bind(sock, (struct sockaddr *)&localAddr, sizeof(localAddr)) == -1) {
    perror("bind");
    return;
}
```

C / C++ UDP/IP Socket Implementation

STEP 3: TRANSMIT DATA

* **Write the data to be sent into a buffer**
* **Allocate an internet address structure to contain destination IP address and port**
* **Transmit the data by calling sendto() function with data buffer and internet address structure as arguments**
* **Note that unlike TCP/IP the size of the data need not be transmitted, this is because datagram delivery semantics ensure the entire buffer will be delivered as a unit**

C / C++ UDP/IP Socket Implementation

```
int BUFFERLEN = 255;
char buf[BUFFERLEN];                    // Allocate buffer for data
sprintf(buf, "hello!");                 // Write data into the buffer

struct sockaddr_in destAddr; // The address/port of the remote endpoint
bzero((char *)&destAddr, sizeof(destAddr));   // zero out allocated memory
destAddr.sin_family = PF_INET;                 // Use Internet addresses
destAddr.sin_addr.s_addr = inet_addr("10.25.43.9");
destAddr.sin_port = htons(13214);

// Send data to the specified destination
if (sendto(sock, buf, strlen(buf) + 1, 0,
(struct sockaddr *)&destAddr, sizeof(destAddr)) != strlen(buf)) {
    perror("sendto");
    return;
}
```

C / C++ UDP/IP Socket Implementation

STEP 4: RECIEVE DATA

* Allocate a buffer to put received data in
* Allocate an internet address structure to hold the sender's IP address and port number
* call recvfrom() function with data buffer and internet address structure as arguments

C / C++ UDP/IP Socket Implementation

```
int BUFFERLEN = 255;
char buf[BUFFERLEN];                          //  Buffer for incoming data

struct sockaddr_in srcAddr;                   // The address/port of sender
bzero((char *)&destAddr, sizeof(srcAddr));  // zero out allocated memory

// Receive data sent to the UDP port
if (recvfrom(sock, buf, sizeof(buf), 0,
(struct sockaddr *)&srcAddr, sizeof(srcAddr)) == -1) {
    perror("recvfrom");
    return;
}
// Sender's address stored in srcAddr structure
```

C / C++ UDP/IP Socket Implementation

STEP 5: CLOSE THE SOCKET

* Remember, there is no connection to close
* However, the socket should still be closed in order to free resources that are no longer needed
* Other hosts have no way of knowing that the connection has been closed
* code ==> close(sock);

C / C++ UDP Broadcasting Socket Implementation

* UDP broadcasting is identical to UDP/IP unicast with two exceptions
  1) The destination address must be set to the broadcast pseudo IP address

     ```
     destAddr.sin_addr.s_addr = inet_addr("255.255.255.255")
     ```

  2) Before data can be broadcast on a socket the application must register its intent to do so

     ```
     int one = 1;
     setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &one, sizeof(one));
     ```

* SO_BROADCAST is a state variable, remains in force until changed
* UDP sockets can receive both unicast and broadcast packets

C / C++ Multicasting Socket Implementation

* **TO TRANSMIT DATA:**

- Multicast transmission is nearly identical to UDP/IP. Make sure the packets are sent to a multicast address
- The SO_BROADCAST option need not be set
- Can set the Time To Live field as shown below

```
unsigned char ttl = 31;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

C / C++ Multicasting Socket Implementation

* **TO RECEIVE DATA:**

- The application must subscribe the socket to a multicast address
- Subscribing to a multicast address is accomplished by calling setsockopt() with the IP_ADD_MEMBERSHIP option

```
struct ip_mreq joinAddr;

// Specify the multicast address to join
joinAddr.imr_multiaddr = inet_addr(“245.8.2.58”);

// Specify which local IP address will do the multicast join
joinAddr.imr_interface = INADDR_ANY;

setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &joinAddr, sizeof(joinAddr))
```

C / C++ Multicasting Socket Implementation

* **TO RECEIVE DATA cont:**

- To cancel a multicast subscription call setsockopt() with the IP_DROP_MEMBERSHIP option

```
struct ip_mreq joinAddr;

// Specify the multicast address to drop
joinAddr.imr_multiaddr = inet_addr(“245.8.2.58”);

// Specify which local IP address will do the multicast drop
joinAddr.imr_interface = INADDR_ANY;

setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &joinAddr, sizeof(joinAddr))
```