**0ssSockets**

|  |  |
| --- | --- |
| **Client Server**  So far, we’ve covered methods for processes to communicate to each other through various means (e.g. files, pipes, named pipes, shared memory). In some cases, it can be beneficial to communicate over a network to processes on other machines. To do this, we must utilize sockets.  As with some of the methods described before, we will be using a client-server design for **network sockets**. The figure on the left illustrates this with multiple client processes on the left writing through the network to a server process on the right. The conduits through which the processes write (i.e., the blue arrows) are created with sockets. Much like with files, clients will **write** messages to the socket and the server will **read** those messages from it.  Socket connections require both processes to have their own sockets. Communication can then be made from a local socket to a remote socket. To enable this communication, a socket must be created, an address defined, and a connection made. Then messages can be sent and received, and the sockets can be ultimately closed.  **Web Browser Example**  To illustrate a client process, these notes will guide you through the above steps by constructing a rudimentary web browser client (the orange box in the image on the right). We will implement our program to connect to an established web server over the Internet. This example can easily be modified to allow communication between processes on the same machine.  **Local Sockets**  Sockets can be utilized for inter-process communication in a similar manner as named pipes. Instead of identifying hosts over a network, file addresses are specified as the socket address and two processes can communicate by connecting to the same file address. | **Socket Communication Over a Network**    **Web Browser**  Web Browser  Process  Internet  Your Computer  Web Server  Process  cs.utsa.edu  port 80  **Socket Communication within a Single Machine**  Process 2  Process 1  Paper  File  Ad |
| **IP Address**  When creating remote connections over a network, we must utilize unique identifiers to specify what server our process will connect to. This is similar to how paths specify the files a process may open. For network access, these identifiers are Internet Protocol (IP) addresses.  IP addresses exist in two formats: 32-bit IPv4 addresses (e.g., 192.168.1.1) and 64-bit IPv6 addresses (e.g., 2001:db8:0:1234:ffff:c0a8:101). For the examples in these notes, we will utilize IPv4 (which is most commonly used). When using these addresses at the socket level, they must first be translated from their human-readable forms to network address form (i.e., binary).  **Port**  If the IP address identifies the remote machine to which our process will connect, a **port** provides a known identifier on the remote machine which both the client and server agree to communicate over. Ports are specified by a 16-bit unsigned integer.  Port numbers between 0 and 1023 are considered system ports and are reserved for well-known protocols and services. When creating your own application, it is wise not to use these ports (or other heavily-utilized ports).  A list of reserved ports can be found at the Internet Assigned Numbers Authority (IANA) registry web site: <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml> | **Some Well-Known Ports**   |  |  | | --- | --- | | **Port Number** | **Description** | | **20** | File Transfer Protocol (FTP) | | **22** | Secure Shell (SSH) | | **23** | Telnet | | **25** | Simple Mail Transfer Protocol (SMTP) | | **80** | Hypertext Transfer Protocol (HTTP) | | **88** | Kerberos | | **123** | Network Time Protocol (NTP) | | **389** | Lightweight Directory Access Protocol (LDAP) | | **443** | HTTP over TLS/SSL (HTTPS) | | **3306** | MySQL | | **8080** | HTTP alternative | |
| **Steps for a Client**  1. Use **socket**(*domainFamily*, *type*, 0) to create the socket as a file descriptor, sd.  2. Use **connect**(sd, *socketAddr*, sizeof(*socketAddr*)) to connect the socket to the server.  3. Send data or a request using  **send**(sd, &*request*, *requestLen*, *flags*)  4. Receive a response from the server using  **recv**(sd, &*response*, sizeof(*response*), *flags*)  which also returns the length of the response.  5. When finished with requests, the client should **close**(sd).  The details of each of the client steps are shown below. We will then show the details of steps for a server. | **Steps for a Server**  1. Use **socket**(*domainFamily*, *type*, 0) to create the socket as a file descriptor, sd.  2. Bind the socket to an address using  **bind**(sd, *socketAddr*, sizeof(*socketAddr*))  3. Use **listen**(sd, 5) to allow clients to connect to the server.  4. Accept a connection from a client using  **accept**(sd, &*socketAddr*, &*socketAddrLen*), receiving commSd for communicating with the client. Through the parameter list, we also receive the client's socket address and the length of that address. **accept** blocks until a client connects.  5. Receive a request from the client using  **recv**(commSd, &*request*, sizeof(*request*), *flags*)  which also returns the length of the request.  6. Write a response to the client using  **write**(commSd, &*response*, *responseLen*)  7. If the server needs to process more requests from this client, it should repeat steps 5 and 6. When communication with the client is complete, the server should **close**(commSd).  8. The server should continue accepting requests from clients by repeating steps 4 thru 7. When it no longer is accepting requests, it should **close**(sd). |
| **Creating a Socket**  int socket(int domain, int type, int protocol)   * Creates a new socket which can be operated on * Returns a file descriptor on success or -1 on error   **Domain** argument specifies a protocol family to be used for communication. The different domains are defined in <sys/socket.h>. Based on the family chosen, the address of the remote connection will be specified differently. In most cases, you will either be utilizing the AF\_INET family for network communication or the AF\_UNIX family for local machine communication. These families utilize IP addresses and file paths for their address formats respectively.  Socket **types** specify communication semantics. The two most commonly used are SOCK\_STREAM and SOCK\_DGRAM. SOCK\_STREAM deals with continuous streams of bytes/characters (similar to files or pipes). SOCK\_DGRAM is for reading fixed-length chunks of data.  The **protocol** should generally be determined by the socket type. By setting it to 0, the most appropriate protocol is chosen. TCP is used for SOCK\_STREAM and UDP is used for SOCK\_DGRAM.  More families and types are available. You can read about them in the socket man page.  **Processes cannot communicate with each other unless both are utilizing sockets of the same family and type.** | **Address Domain Families (sys/socket.h)**   |  |  | | --- | --- | | **Domain** | **Purpose** | | **AF\_UNIX, AF\_LOCAL** | Local communication | | **AF\_INET** | IPv4 protocols | | **AF\_INET6** | IPv6 protocols |   **Socket Types (sys/socket.h)**   |  |  | | --- | --- | | **Type** | **Purpose** | | **SOCK\_STREAM** | Provides sequenced, reliable, two-way, connection-based byte streams. | | **SOCK\_DGRAM** | Supports datagrams (connectionless, unreliable messages of fixed maximum length). | | **SOCK\_SEQPACKET** | Provides sequenced, reliable, two-way, connection-based byte streams. | | **SOCK\_RAW** | Provides raw network protocol access. | | **SOCK\_RDM** | Provides a reliable datagram layer that does not guarantee ordering. |   **Example 1-1:** Create a stream socket to be used over the Internet.  #include <stdio.h>  #include <sys/socket.h>  void errExit(const char szFmt[], ... ); // prototype  int main()  {  int sd; // socket descriptor    // create the socket with the appropriate properties  sd = socket(AF\_INET, SOCK\_STREAM, 0);  // verify socket  if (sd == -1)  errExit("Failed to create socket”);  return 0;  } |
| **Defining a Host**  With a socket created, we must now define a remote host before actually making a connection. To do this, we utilize the sockaddr structure as defined in <netinet/in.h>.  Different address families utilize a different sockaddr structure. When we create the connection, the sockaddr struct we created is cast to type struct sockaddr. This is polymorphism. To access the fields in the original structure, the sockaddr must be downcast before dereferencing.  The example in these notes uses the AF\_INET socket family to access remote processes over the network/Internet. By using the AF\_UNIX or AF\_LOCAL socket family, we could easily define a process on the local machine instead.  **Converting IP Addresses**  int inet\_pton (int af, const char \*src, void \*dst)   * Converts the character string src into a network address structure in the af address family, then copies the network address structure to dst. * Useful for converting IP addresses to uint32\_t for in\_addr * Returns 1 on success, 0 on invalid address, or -1 on invalid address family * Available in <arpa/inet.h>   **Converting to Network Byte Order**  uint16\_t htons (uint16\_t hostshort)   * Converts from host byte order to network short byte order * Useful for converting ports to uint16\_t for sin\_port * This helps avoid issues with little/big endian integer incompatibility.   Although not shown in these notes, if you have a URL, you can use gethostbyname(*url*) to help construct the socket address. | **Relevant Socket Address Structs (netinet/in.h)**  // Generic socket address. When binding a connection,  // the appropriate sockaddr will be cast to this struct type  struct sockaddr {  sa\_family\_t sa\_family;  char sa\_data[14];  };  // AF\_INET socket address  struct sockaddr\_in {  sa\_family\_t sin\_family; // socket family (AF\_INET)  in\_port\_t sin\_port; // port number, network byte order  struct in\_addr sin\_addr; // host address (see below)  };  // AF\_UNIX/AF\_LOCAL socket address  struct sockaddr\_un {  sa\_family\_t sun\_family; // socket family (AF\_UNIX)  char sun\_path[108]; // pathname  };  // AF\_INET6 socket address  struct sockaddr\_in6 {  u\_int16\_t sin6\_family; // address family (AF\_INET6)  u\_int16\_t sin6\_port; // port number, network byte order  u\_int32\_t sin6\_flowinfo; // IPv6 flow information  struct in6\_addr sin6\_addr; // IPv6 address  u\_int32\_t sin6\_scope\_id; // Scope ID  };  struct in\_addr {  uint32\_t s\_addr; // address in byte order  };  struct in6\_addr {  unsigned char s6\_addr[16]; // IPv6 address  }; |
| **Connecting**  With a socket and sockaddr structure to connect to, a connection can now be made with the connect function.  int connect (int sockfd, const struct sockaddr \*addr, socklen\_t addrlen)   * Connects the socket referred to by the file descriptor **sockfd** to the address specified by **addr**. * The **addrlen** specifies the size of addr. * Returns 0 on success or -1 on error   To test the ability to connect, we will connect to a remote webserver on over the Internet. By default, http requests are made over port 80.  **Connections and Other Types of Sockets**  The notion of a “connection” is relevant to TCP sockets (SOCK\_STREAM). By establishing a TCP connection, we have created a private, *reliable* stream with which to transmit data (generally with packets). Other sockets types, such as UDP (SOCK\_DGRAM), are simpler in that they do not keep track of state and there is no concept of a “connection”. This allows for faster communication, but no guarantee about the data reaching its destination.  You will learn more about these protocols in Operating Systems and Networks. | **Example 1-2:** Connect to a remote host over the Internet  #include <stdio.h>  #include <sys/socket.h>  #include <netinet/in.h> // sockaddr structs  #include <arpa/inet.h> // for IP address conversions  void errExit(const char szFmt[], ... ); // prototype  #define REMOTE\_HOST "129.115.28.4" // cs.utsa.edu  int main()  {  int sd; // socket descriptor  struct sockaddr\_in host; // remote host to connect to  int connectStatus;  // create the socket with the appropriate properties  sd = socket(AF\_INET, SOCK\_STREAM, 0);  // verify socket  if (sd == -1)  errExit("Failed to create socket");    // set up sockaddr  host.sin\_family = AF\_INET;  host.sin\_port = htons(80); // http  inet\_pton(AF\_INET, REMOTE\_HOST, &(host.sin\_addr.s\_addr));    // Connect to remote server  connectStatus = connect(sd, (struct sockaddr \*) &host,  sizeof(host));  if (connectStatus < 0)  errExit("Failed to connect");    printf("Connection Established!");    return 0;  }  Why are we able to connect to this server even though we didn’t implement it?  ?? |
| **Sending Data**  After a connection has been established over the socket descriptor, data can be sent to it in a similar way as writing to a file using the send() function. You can actually use write() to send data to a socket as well.  ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);   * Writes data to a connected socket (and thus transmits the data to another socket). * sockfd is the socket descriptor created with socket() * buf is a constant, void pointer so it can be any structure or character buffer. This is the data that will be transmitted. * len is the size of the message to be transmitted. The size\_t type is an unsigned type used to represent size in bytes. The sizeof operator returns the appropriate size. * There are multiple flags that can be passed to dictate the behavior of the message (e.g., non-blocking, end of record, etc). These flags can be found in the send(2) man page. For now, we will not pass any flags by setting flags to 0. * Returns the number of bytes sent on success or -1 on error .   **Sending a Request to a Web Server**  In Example 1.3, we will send the remote webserver to which we have connected a GET request. For Hypertext Transfer Protocol (HTTP), messages designated as GET *request* data. This is what happens in your web browser when you type in a url to navigate to a website. | **Example 1-3:** Make a request to a remote host  #include <stdio.h>  #include <sys/socket.h>  #include <netinet/in.h> // sockaddr structs  #include <arpa/inet.h> // for IP address conversions  #include <string.h> // strlen()  void errExit(const char szFmt[], ... ); // prototype  #define REMOTE\_HOST "129.115.28.4" // cs.utsa.edu  int main()  {  int sd; // socket descriptor  struct sockaddr\_in host; // remote host to connect to  int connectStatus;  ssize\_t bytesSent;  char \*request;    // create the socket with the appropriate properties  sd = socket(AF\_INET, SOCK\_STREAM, 0);    // verify socket  if (sd == -1)  errExit("Failed to create socket");    // set up sockaddr  host.sin\_family = AF\_INET;  host.sin\_port = htons(80); // http  inet\_pton(AF\_INET, REMOTE\_HOST, &(host.sin\_addr.s\_addr));    // Connect to remote server  connectStatus = connect(sd, (struct sockaddr \*) &host,  sizeof(host));  if (connectStatus < 0)  errExit("Failed to connect");    printf("Connection Established!\n");    // make the request  request = "GET / HTTP/1.0/\r\n\r\n";  bytesSent = send(sd, request, strlen(request) + 1, 0);  if(bytesSent != strlen(request) + 1)  errExit("Failed to send request");    printf("Request Sent!\n");    return 0;  }  Why do we add 1 to strlen()?  ?? |
| **Receiving Data**  Data must be read from a socket in a similar way to reading from a file. This is done with the recv() function. As with files, you can also use read() to read data from a socket.  ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);   * Reads data from a connected socket. * sockfd is the socket descriptor created with socket() * buf is a void pointer so it can be any structure or character buffer. This is the buffer to which the data will be read. * len is the size of the message to be read. The size\_t type is an unsigned type used to represent size in bytes. The sizeof operator returns the appropriate size. * There are multiple flags that can be passed to dictate the behavior of the read (e.g., non-blocking, close-on-exec, etc). These flags can be found in the recv(2) man page. For now, we will not pass any flags by setting flags to 0. * Returns the number of bytes read on success or -1 on error. If the remote socket performs and orderly shutdown, the return value will be 0.   **Web Server Response**  So far, we have set up a connection and sent a request to a remote server. Typically, in client-server architectures, such requests are followed by a response from the server. In our http example, the GET request should invoke a response in the form of an HTML web page.  Note, that since we are simply dealing with the text messages between the client and the server, we will not be rendering the HTML response into an actual web page. Instead, we will simply view the markup to demonstrate the connection. Furthermore, since the request buffer is limited to 2048 in the example, that is the maximum amount of a response that will be read (web pages are typically larger than this).  **Closing the Socket**  As with files, the socket must be closed. The close() function available in unistd.h is sufficient.  int close(int fd);   * Closes a file descriptor. * Returns 0 on success, -1 on error. | **Example 1-4:** Receive a response from the remote host  #include <stdio.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <string.h>  #include <unistd.h> // close()  void errExit(const char szFmt[], ... ); // prototype  #define REMOTE\_HOST "129.115.28.4" // cs.utsa.edu  int main()  {  int sd; // socket descriptor  struct sockaddr\_in host; // remote host to connect to  int connectStatus;  ssize\_t bytesSent;  ssize\_t bytesReceived;  char \*request;  char response[2048];    // create the socket with the appropriate properties  sd = socket(AF\_INET, SOCK\_STREAM, 0);    // verify socket  if (sd == -1)  errExit("Failed to create socket");    // set up sockaddr  host.sin\_family = AF\_INET;  host.sin\_port = htons(80); // http  inet\_pton(AF\_INET, REMOTE\_HOST, &(host.sin\_addr.s\_addr));    // Connect to remote server  connectStatus = connect(sd, (struct sockaddr \*) &host,  sizeof(host));  if (connectStatus < 0)  errExit("Failed to connect");    printf("Connection Established!\n");    // make the request  request = "GET / HTTP/1.0/\r\n\r\n";  bytesSent = send(sd, request, strlen(request) + 1, 0);  if(bytesSent != strlen(request) + 1)  errExit("Failed to send request");    printf("Request Sent!\n");  // read reply  bytesReceived = recv(sd, response, sizeof(response), 0);  if(bytesReceived < 0)  errExit("Failed to receive response");  printf("Response:\n%s\n", response);    close(sd);  return 0;  } |
| The completed web client code can be found at  **/usr/local/courses/rslavin/cs3423/sockets/webClient.c** | $ gcc -o webClient webClient.c  $ ./webClient  Connection Established!  Request Sent!  Response:  HTTP/1.1 200 OK  Date: Thu, 02 Nov 2017 17:26:52 GMT  Server: Apache/2.2.3 (Debian) PHP/5.2.12-0.dotdeb.0 with Suhosin-Patch mod\_ssl/2.2.3 OpenSSL/0.9.8c  X-Powered-By: PHP/5.2.12-0.dotdeb.0  Set-Cookie: CMSSESSID09551078=f0ced552671c3a89298d5b759c71040c; path=/  Expires: Thu, 19 Nov 1981 08:52:00 GMT  Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0  Pragma: no-cache  Connection: close  Content-Type: text/html; charset=UTF-8  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" >  <head>  <title>Department of Computer Science</title>  <base href="http://www.cs.utsa.edu/" />  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />  <link rel="shortcut icon" href="/uploads/images/favicon.ico" type="image/x-icon">  <link rel="stylesheet" type="text/css" media="screen" href="http://www.cs.utsa.edu/stylesheet.php?cssid=42&amp;mediatype=screen " />  <link rel="stylesheet" type="text/css" href="http://www.cs.utsa.edu/stylesheet.php?cssid=48" />  <!-- TinyMCE Session vars empty --></head>  <body class="primary">  <div id="header">  <div class="head">  <div id="newutsaheader" class="head">The University of Texas at San Antonio</div>  <div id="utsaheader" class="head">  <div id="1" class="head">Department of Computer Science</div>  </div>  Why does the response not include the entire web page’s html?  ?? |
| **Socket Server**  To complete our client-server architecture, we must implement our own server. In the above example, the server was an established web server created by someone else. Since our socket utilized the AF\_INET family, we were able to create a connection with the remote server and make requests. For our own implementation, our server (the orange box on the right) should be able to listen for requests on a designated port and respond to them. | **Web Browser**  Server  Process  Computer B  Client  Process 1  Computer A  Network  Client  Process 2  Computer B  agreed port |
| **Creating a Socket for Queuing Connection Requests**  In order for a connection to be made, both the client and server processes must create their own sockets. So just as with the client, our server process must begin by utilizing the socket() function with the appropriate parameters.  Since it is possible for multiple connection requests to be made to a single server socket, we will set up an initial socket for queuing those requests and then spawn secondary sockets for the actual communication that is made once a connection is fully established. | **Example 2-1:** Create a stream socket for listening on the server.  #include <stdio.h>  #include <sys/socket.h>  void errExit(const char szFmt[], ... ); // prototype  int main()  {  int sd; // socket descriptor    // create the socket with the appropriate properties  sd = socket(AF\_INET, SOCK\_STREAM, 0);  // verify socket  if (sd == -1)  errExit("Failed to create socket”);    return 0;  } |
| **Binding**  Our server process must be able to “listen” for incoming connections. Recall that our client used the connect() function to establish a connection to the remote host with its socket. Since the server could not know exactly when we make that connection, it must continually listen to the agreed-upon address and port. We assign these properties through an action called **binding**. This is similar to how the client uses connect() to provide the remote address and port and establish a connection, only bind sets up the socket for listening.  int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrLen);   * Assigns a local address to the socket to listen on. * Connects the socket referred to by the file descriptor **sockfd** to the address specified by **addr**. * The **addrLen** specifies the size of addr. * Returns 0 on success or -1 on error   A sockaddr structure must be defined with the appropriate attributes for the connection. Since we want our server to listen on all available local interfaces/addresses, s\_addr should be set to INADDR\_ANY.  **Note**: our server process will operate on a port other than port 80 so it does not interfere with any web server that may already exist on the machine. This means that we will need to update webClient.c to operate on the same port. We will use port 8088. | **Example 2-2:** Binding a socket.  #include <stdio.h>  #include <sys/socket.h>  #include <netinet/in.h> // sockaddr structs  void errExit(const char szFmt[], ... ); // prototype  int main()  {  int sd; // socket descriptor  struct sockaddr\_in local; // local host  int bindStatus;    // create the socket with the appropriate properties  sd = socket(AF\_INET, SOCK\_STREAM, 0);  // verify socket  if (sd == -1)  errExit("Failed to create socket”);  // set up sockaddr  local.sin\_family = AF\_INET;  local.sin\_port = htons(8088);  local.sin\_addr.s\_addr = INADDR\_ANY; // all available interfaces    // bind the address  bindStatus = bind(sd, (struct sockaddr \*) &local, sizeof(local));  if (bindStatus < 0)  errExit("Failed to bind socket");  printf("Socket Bound!\n");  return 0;  } |
| **Listening and Accepting**  With the socket bound, it now needs to be set to listen for incoming connections from the client. Then, when a connection attempt is made by a client to the socket, the process can accept and establish the connection.  int listen(int sockfd, int backLog);   * Sets the socket, **sockfd**, to passive mode to be used for incoming connection requests. * When multiple connections are made a queue is created with those requests. The **backlog** parameter signifies a *hint* about the maximum length of that queue. This value is often ignored and should not be relied upon. If a connection request is made by a client and the queue is full, the client received the ECONNREFUSED response. * Returns 0 on success or -1 on error   Once the socket is in listen mode, connection requests can be made to the socket from clients (filling the backlog queue) and the server can begin accepting those requests. When a request is accepted, a *new* socket is created through which communication is made. Effectively, the first socket exists solely for queueing requests for which new, secondary sockets are created for the actual communication. The accept() function takes these requests and generates the communication socket.  int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrLen);   * Extracts the first connection request from the queue for **sockfd** and returns a new socket descriptor for communication. * **addr** is a pointer to a sockaddr structure in which to store the client’s sockaddr. * **addrLen** is a value-result argument which should be initialized to the size (in bytes) of the structure pointed to by addr. On return, it will be filled with the actual size of the address structure. * If there are no connection request in the queue, accept() will block until one is made. * Returns a file descriptor to the new socket on success or -1 on error | **Example 2-3:** Configuring a socket to listen and accept requests.  #include <stdio.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <unistd.h> // close  void errExit(const char szFmt[], ... ); // prototype  int main()  {  int sd; // socket descriptor  int commSd; // communication socket descriptor  struct sockaddr\_in local; // local host  struct sockaddr\_in client; // remote client  int clientSaSize; // client sockaddr size  int bindStatus;    // create the socket with the appropriate properties  sd = socket(AF\_INET, SOCK\_STREAM, 0);  // verify socket  if (sd == -1)  errExit("Failed to create socket");  // set up sockaddr  local.sin\_family = AF\_INET;  local.sin\_port = htons(8088);  local.sin\_addr.s\_addr = INADDR\_ANY; // all available interfaces    // bind the address  bindStatus = bind(sd, (struct sockaddr \*) &local, sizeof(local));  if (bindStatus < 0)  errExit("Failed to bind socket");  printf("Socket Bound!\n");  // set socket to listen mode  listen(sd, 1);  printf("Listening for connection requests...\n");  // accept a request from the queue and create a new socket  // Note that accept will loop until a request is in the queue  clientSaSize = sizeof(struct sockaddr\_in);  commSd = accept(sd, (struct sockaddr \*) &client,  (socklen\_t \*) &clientSaSize);  // verify socket  if (commSd == -1)  errExit("Failed to create communication socket");  printf("Connection Established!\n");  close(sd);  close(commSd);  return 0;  } |
| **Testing a Connection**  Now that our server can listen for and establish connections with clients, we can test it. Open two terminals (or two machines on the same network), one with the compiled client and the other with the compiled server.  Run the server. It should print that it is listening for connection requests. At this point, the server continually listens while accept() waits for the request queue to receive a request. Next, run the client. It should immediately make a request to your server, causing the server to generate a communication socket and establish a connection.  **Note:** Be sure to modify client.c to include the host’s IP address and the new port (8088). | **Example 2-4:** Testing a connection  # Terminal 1: Server  $ gcc -o server server.c  $ ./server  Socket Bound!  Listening for connection requests... (it then waits for a connection request)  # Terminal 2: Client  $ gcc -o client client.c  $ ./client  Connection Established!  Request Sent! (the server accepts the connection here)  # Terminal 1: Server (continued from above)  Connection Established! (at this point, the server process closes the connection) |
| **Responding**  Now that a connection is established on its own socket, communication can be made between the client and the server (instead of immediately closing the connection).  Much like with file descriptors, we can write to a socket with the write() function. We can also read the request sent from the client in the same way we read the response from the server with recv().  The code on the left reads the request from the client immediately after accepting the connection. Note that it is reading from commSd (the communications socket) and not sd (the request queue socket). Then the server uses write() to send a message back to the client. Both processes can read *and* write to the same socket since it was opened with the SOCK\_STREAM type, which allows for two-way communication. | **Example 3-1:** Communication.  #include <stdio.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <unistd.h> // close  void errExit(const char szFmt[], ... ); // prototype  int main()  {  int sd; // socket descriptor  int commSd; // communication socket descriptor  struct sockaddr\_in local; // local host  struct sockaddr\_in client; // remote client  int clientSaSize; // client sockaddr size  int bindStatus;  char \*message; // message for client  char msgFrClient[1024]; // message from client  int bytesReceived; // size of message from client    // create the socket with the appropriate properties  sd = socket(AF\_INET, SOCK\_STREAM, 0);  // verify socket  if (sd == -1)  errExit("Failed to create socket");  // set up sockaddr  local.sin\_family = AF\_INET;  local.sin\_port = htons(8088);  local.sin\_addr.s\_addr = INADDR\_ANY; // all available interfaces    // bind the address  bindStatus = bind(sd, (struct sockaddr \*) &local, sizeof(local));  if (bindStatus < 0)  errExit("Failed to bind socket");  printf("Socket Bound!\n");  // set socket to listen mode  listen(sd, 1);  printf("Listening for connection requests...\n");  // accept a request from the queue and create a new socket  // Note that accept will loop until a request is in the queue  clientSaSize = sizeof(struct sockaddr\_in);  commSd = accept(sd, (struct sockaddr \*) &client,  (socklen\_t \*) &clientSaSize);  // verify socket  if (commSd == -1)  errExit("Failed to create communication socket");  printf("Connection Established!\n");  // read message from client  bytesReceived = recv(commSd, msgFrClient, 1024, 0);  if(bytesReceived > 0)  printf("Message from client:\n%s\n", msgFrClient);  else  printf("No message from client.\n");  // send a message  message = "Hello from the server!";  write(commSd, message, strlen(message) + 1);  close(sd);  close(commSd);  return 0;  } |
| **Two-way Communication**  With the server complete with both reads and writes over the sockets, we can test client-server communication. Example 1-10 demonstrates how client.c and server.c communicate with each other. By modifying the messages and/or the socket type, this can easily be extended to share any type of information.  A modified version of client.c which takes arguments for the remote server and message to send as a message can be found at **/usr/local/courses/rslavin/cs3423/sockets/client.c** | **Example 3-2:** Testing a connection  # Terminal 1: Server  $ gcc -o server server.c  $ ./server  Socket Bound!  Listening for connection requests... (it then waits for a connection request)  # Terminal 2: Client  $ gcc -o client client.c  $ ./client  Connection Established!  Request Sent! (the server accepts the connection here)  # Terminal 1: Server (continued from above)  Connection Established! (at this point, the server process closes the connection)  Message from client:  GET / HTTP/1.0/  # Terminal 2: Client (continued from above)  Response:  Hello from the server! |
| **Continually Accepting Connections**  Generally, servers do not simply wait for a connection, read and write, and then close. Ideally, our server would run continually and accept multiple connections from various clients without exiting. Our implementation can be improved by enclosing the accept() in a while loop so it will continually accept connections from the queue.  The completed server code can be found at  **/usr/local/courses/rslavin/cs3423/sockets/server.c** | #include <netinet/in.h>  #include <unistd.h> // close  void errExit(const char szFmt[], ... ); // prototype  int main()  {  int sd; // socket descriptor  int commSd; // communication socket descriptor  struct sockaddr\_in local; // local host  struct sockaddr\_in client; // remote client  int clientSaSize; // client sockaddr size  int bindStatus;  char \*message; // message for client  char msgFrClient[1024]; // message from client  int bytesReceived; // size of message from client    // create the socket with the appropriate properties  sd = socket(AF\_INET, SOCK\_STREAM, 0);  // verify socket  if (sd == -1)  errExit("Failed to create socket");  // set up sockaddr  local.sin\_family = AF\_INET;  local.sin\_port = htons(8088);  local.sin\_addr.s\_addr = INADDR\_ANY; // all available interfaces    // bind the address  bindStatus = bind(sd, (struct sockaddr \*) &local, sizeof(local));  if (bindStatus < 0)  errExit("Failed to bind socket");  printf("Socket Bound!\n");  // set socket to listen mode  listen(sd, 1);  printf("Listening for connection requests...\n");  // accept a request from the queue and create a new socket  // Note that accept will loop until a request is in the queue  clientSaSize = sizeof(struct sockaddr\_in);  while((commSd = accept(sd, (struct sockaddr \*) &client,  (socklen\_t \*) &clientSaSize)) != -1){  printf("Connection Established!\n");  // read message from client  bytesReceived = recv(commSd, msgFrClient, 1024, 0);  if(bytesReceived > 0)  printf("Message from client:\n%s\n", msgFrClient);  else  printf("No message from client.\n");  // send a message  message = "Your message was received!";  write(commSd, message, strlen(message) + 1);  }  close(sd);  close(commSd);  if(commSd == -1)  errExit("Failed to create communication socket");  return 0;  } |

**©2018 Rocky Slavin and Larry W. Clark,** UTSA CS students may make copies for their personal use