Daytime Client/Server

CIS-3152, Spring 2014
Peter C. Chapin
Addresses

• IP Addresses are assigned to interfaces
  • A machine with multiple interfaces gets multiple addresses.
    – Interfaces can be physical or virtual.
• “Machine's IP address” is technically incorrect.
  • *Machines don't have addresses, interfaces do.*
  • However, many machines have only one (relevant) interface.
    – Thus talk of a “machine's IP address” is common.
Connections

- TCP connections are described by a 4-tuple
  - \((\text{src address}, \text{src port}, \text{dest address}, \text{dest port})\)
Connections (Continued)

• TCP connections are bi-directional
  • Words like “source” and “destination” don't apply!
  • IP packets have sources and destinations, but on a connection both endpoints can send or receive data.

• However: creating a connection is asymmetric
  • Client active: initiates connection (dials the phone)
  • Server passive: accepts the connection (picks up the phone when it rings)
Connections (Continued)

- *Once connection is established, peers equal.*
- Either side can initiate a shutdown.
  - The first side that closes the connection does an **active close**.
  - The other side responds with a **passive close**.
- Which side does the active close is an application level decision.
  - Either side must be prepared for the other to close unexpectedly.
    - You have no idea what your peer will do.
Sockets is Protocol Independent

• **Important!**
  • The sockets API is *not specific to TCP/IP*
  • On machines supporting multiple protocols (OSI, IPX/SPX, etc) sockets could potentially work with all of them.
    - We will care about this when we look at IPv6.

• Design of Sockets is object oriented!
  • But... since C is not an OO language, the interface is somewhat awkward.
  • Knowing this helps explain the awkwardness
    - And helps make it more acceptable!
Unix Style Error Handling

• Unix system calls follow a simple tradition.
  • Call returns integer -1 when error occurs
  • Sets a global integer `errno` with an error code.
    – Consult man page for specific error code possibilities.
      • Usually shown with symbolic name #defined in `<errno.h>`. For example: `EPERM` (meaning permission denied).
  • Check each system call for -1 return.
    – If found, consult `errno` for more specific information.
  • NOTE: Not all system calls follow this approach. Check the man page to be sure.
Quick Error Messages

• Checking `errno` all the time is a pain.

• Library function `perror` simplifies the process of producing useful error messages.
  • Looks up a generic message in an internal table using `errno` value as a table index.
  • EXAMPLE: Suppose `errno` set to `EPERM`
    - `perror("Unable to do operation");`
    - Displays: “Unable to do operation: Permission denied”
  • Sends message to standard error file.

• [Demonstrate `connect` man page]
Daytime Protocol

• A simple protocol good for testing.
  • Can focus on network issues because the protocol is trivial.
  • Stevens uses it as a first example in his book. We will also.

• Described by RFC-867
  • Read it!
    – It's very short... unlike some of the RFCs we'll look at later!

• [Demonstrate the RFC index]
Protocol Overview

- **Daytime protocol steps:**
  - Client connects to server port 13 (default)
  - Server sends ASCII string containing the date and time.
    - One line recommended
    - No particular format is required
  - Server closes the connection (active close)

- **NOTE:**
  - Client need not send any data (anything sent is ignored)
Daytime Client Using Sockets

- **Client steps:**
  - Create a socket object (inside the kernel) to represent the connection's endpoint.
  - Prepare a `sockaddr_in` structure to hold the server address and port.
  - Connect to the server.
  - Read the connection (like a file) until an end-of-file indication appears (that is, loop).
    - Sockets will indicate end-of-file when the server closes the connection.
  - Close the connection.
Create a Socket

• Creating a kernel socket object
  • if ((socket_handle = socket(PF_INET, SOCK_STREAM, 0)) == -1 ) {
      perror(“Unable to create socket”);
      return error_code;
  }

• Include headers as necessary (see man pages)
• socket_handle is an integer file handle
• PF_INET specifies the “INET” protocol family (TCP/IPv4)
• SOCK_STREAM specifies a stream protocol (TCP)
• perror is a library function that simplifies error handling.
Prepare Address Structure

- Fill in a `sockaddr_in` structure.
  ```c
  memset(&server_address, 0, sizeof(server_address));
  server_address.sin_family = AF_INET;
  server_address.sin_port   = htons(port);
  if (inet_pton(AF_INET, argv[1],
                   &server_address.sin_addr) <= 0) {
    fprintf(stderr, "Unable to convert address.\n");
    close(socket_handle);
    return error_code;
  }
  ```
  - Zero structure first to put unused fields into a default state.
  - Use `htons` to convert host to network byte order
  - Use `inet_pton` to convert address from “presentation” to “network” form.
Connect To Server

- Call the connect function.
  - if (connect(socket_handle, (struct sockaddr *) &server_address, sizeof(server_address)) == -1) {
    perror("Unable to connect to server");
    close(socket_handle);
    return error_code;
  }

- You must pass a pointer to the server address structure.
  - But you must cast it into a generic sockaddr pointer first!
  - This is like converting to a base class in C++.
  - connect examines the structure and the socket to figure out what protocol you are trying to use.
Read Server's Data

• Read the data from the server like a file.
  • while ((count = read(socket_handle, buffer, BUFFER_SIZE - 1)) > 0) {
    buffer[count] = '\0';
    fputs(buffer, stdout);
  }

• Repeatedly try to read \texttt{BUFFER\_SIZE - 1} bytes.
• Data may arrive in pieces (one byte at a time even)
• Just read and print (in this case) each piece as it arrives.
• \texttt{read} will block (wait) if no data has arrived.
• \texttt{read} returns zero when connection closed.
Don't Forget Error Handling

• If read fails (due to network failure) the user will want to know.
  • if (count < 0) {
    perror("Problem reading socket");
    close(socket_handle);
    return 1;
  }

• Errors on the network are common
  - Network is orders of magnitude less reliable than memory or disks.

• You must write code to consider these errors.
Code Review

[Demonstrate complete client]
Daytime Server Using Sockets

- Server steps:
  - Create a socket object to represent the listening endpoint.
  - Prepare a `sockaddr_in` structure to specify the server port.
  - Bind the socket to the desired address.
  - Listen on the socket.
  - Accept a connection.
  - Write to the connection (like a file).
  - Close the connection.
  - Loop back and accept the next connection.
Create a Socket

• Exactly the same as with the client.

  • if ((listen_handle = socket(PF_INET, SOCK_STREAM, 0))
      == -1) {
      perror("Unable to create socket");
      return error_code;
  }
Prepare Address Structure

• Similar to the client
  
  ```c
  memset(&server_address, 0, sizeof(server_address));
  server_address.sin_family      = AF_INET;
  server_address.sin_addr.s_addr = htonl(INADDR_ANY);
  server_address.sin_port        = htons(port);
  ```

• Zero out the address structure to give unspecified fields appropriate default values.

• Use `INADDR_ANY` to specify listening on any IP address (any interface).

• Use `htonl` and `htons` to deal with endianness issues in a portable way.
Bind Socket to Address

• Associate the socket with the desired address. This is called “binding.”

  if (bind(listen_handle, (struct sockaddr *) &server_address, sizeof(server_address)) == -1) {
      perror("Unable to bind socket");
      close(listen_handle);
      return error_code;
  }

• Binding fails if, for example:
  - The process does not have permission to use the address/port
  - The address/port is already in use by another server.

• **Binding does not entail any network activity.**
Listen on Socket

• This allows connections to be accepted.
  • if (listen(listen_handle, 32) == -1) {
    perror("Unable to listen");
    close(listen_handle);
    return error_code;
  }

• After `listen`, connections will no longer be refused.

• Second parameter controls size of “backlog” queue.
  – Number of connections that can be created without being accepted.
  – Often ignored. Each OS has its own idea about how to manage this value internally.
Accept Connection

- This is how to actually accept a connection.
  - `client_length = sizeof(client_address);`
  - `connection_handle = accept(listen_handle,`
    - `(struct sockaddr *) &client_address,`
    - `&client_address,`
    - `&client_length);`
  - `if (connection_handle == -1) {`
    - `perror("Accept failed");`
  }

- The `accept` function returns a handle to a new socket representing the connection endpoint.
  - ... different from the listening socket!
- The `accept` function's third parameter is “in/out.”
  - `client_length` must be initialized with size of space.
  - `accept` modifies `client_length` to return used space.
Other Details

- Server reads/writes the connection like a file.
  - Just like the client.
  - If client closes first, server will get end-of-file indication.
- Server closes the connection with `close`.
  - Just like the client.
- Listening socket remains open.
  - Server loops back and calls `accept` again to get the next connection.
Code Review

[Demonstrate complete server]
Iterative Server

• The server described is “iterative.”
  • Only accepts one connection at a time.
  • If a connection arrives while one is being serviced, the new connection is added to the backlog queue.
    - Client making that connection must wait.
    - If current connection takes a “long time” the waiting client won't be happy. Example:
      • Servicing current client is time consuming.
      • Current client is unresponsive.
      • Current client is malicious.

• Inefficient use of resources.
Iterative Daytime Server

• BUT... *iterative servers are easy to implement.*

• Iterative servers are appropriate for some protocols:
  • When service provided is trivial, **AND**
  • When server does not have to wait for client commands, **AND**
  • When server does the active close.

• *Daytime protocol meets these requirements!*
Windows Sockets

- Windows uses “WinSock”, not POSIX sockets.
  - Function names all begin with “WSA”
    - WSAConnect, WSAAccept, etc.
  - Functions have similar purpose to their POSIX counterparts, but very different parameter lists, etc.
  - More complicated to use.
- WinSock needs explicit initialization.
  - ... since it is in a DLL that needs to be loaded.
  - Use WSAStartup and WSACleanup.
- Retrieve error codes with WSAGetLastError
Compatibility Library

- Compatibility library eases porting of Unix programs.
  - Provides functions like `connect`, `accept`, etc with POSIX semantics.
  - Implemented on top of the WSA equivalents.
- Not 100% compatible!
  - Still need to use WSA functions to initialize WinSock, get error codes, etc.
  - Some of the data types are different.
- Consult the MSDN library for the details.