PROJECT 1: A BASIC DISTRIBUTED SYSTEM FACILITY

deadline: *Feb 4, 2002, 12noon*

Introduction

In this project you will implement a distributed system facility, based on CLIENT-SERVER programming. A client submits a request for processing, and "the system" takes care of executing the request. The servers are *not* in well-known locations, but the *location server* or *name server* (NS) is. The client should communicate with the NS process, which perhaps executes in a different machine, to obtain the location of the server process that is able to fulfill the request made by the client. If more than a single server is capable of fullfiling the request, you can use any mechanism to choose a server.

Processes communicate via the "socket" mechanism. Sockets are used to pass data from one machine to another. Although sockets can be of two types, namely connection-oriented (CO) and connectionless (CL), you are free to choose the type of sockets you use, as long as the program works 100% of the time. The client will make a request, "the system" will process it, and display the time of receiving the request, the response and the time it displays the response.

In this project, server processes register with the location server, advertising their port numbers and machines they are executing in. The service should only be available after the server has registered with the location server.

The interface of your program should be simple yet effective (a prompt or a visual interface are both acceptable). The service will be to consult a database, which, for this project, you can assume it is a file in AFS.

Requirements

The clients and servers will be run possibly in different machines; you can assume that the location server runs in a specific machine (i.e., you can hardwire/hardcode the location of this server, but remember that other students in the class are also running similar programs perhaps in the same machine).

Each server will fork a different process to take care of the specific request. When the server process finishes processing the request, it can simply exit or it can wait for the next request that arrives. The number of server processes can be dynamically decided, depending on the load and the load fluctuation. This means that your program should be able to handle several simultaneous requests to the same service.

The server processes need not actually execute the process, but they need to be able to register with the LS. After getting a request from a remote client, they may invoke another process/program to execute the request.

Technical Details

The different processes will exchange data through the communication facilities provided (e.g., sockets). Since there are two types of socket communication (CO and CL), request will probably have different response times. You should choose the best design for your system, and justify it.

An example of the socket interface is provided by UNIX can be found in `/afs/pitt.edu/usr36/mosse/public/cs2511/socket` in `/afs/pitt.edu/usr36/mosse/public/cs2511/client-server.intro` there is a small introduction to client/server paradigm. Feel free to copy, understand, and peruse the directory. The program was compiled on Ultrix (wow-that-old?), so it may be slightly different in the Solaris, Linux, or other architectures. Consult the `man` pages. Another excellent source of information is the Stevens book on Networking or web tutorials.

Sockets

Used by processes to communicate with each other on same machine or across the internet; the example below is for connection-oriented.

A typical scenario allows the server to start first, listen for clients to start and attempt communication. Some structures needed to use sockets are defined in `<netinet/in.h>`, `<sys/socket.h>`, and `<sys/un.h>`.

In the connection oriented protocol, the following is the handshaking protocol (in macro steps):
mrcurrsizerm

```
SERVER                                          CLIENT
1- socket()                                     socket()
2- bind()                                       /* client must call socket() before calling connect() */
3- listen()
4- accept() /* blocks until connection from client */
                                                5- connect() /* blocks until connected to server */
                                                6- send()
7- receive() /* blocks until msg arrives */
8- process request
9- send()
                                                /* data reply */
                                                10- receive() /* can also set a timeout value
```

Below find some of the descriptions of the system calls used with sockets.

```
int socket  (  int family,               /* can be AF_UNIX (same machine) or AF_INET */
               int type,                 /* can be SOCK_STREAM or SOCK_DGRAM */
               int protocol              /* is typically 0 */
)   /* system call return a small int value, analogous to a file descriptor. */
```

```
int bind  (    int sock_desc,            /* value returned by the socket() call */
               struct sockaddr *myaddr,  /* ptr to protocol specific address */
               int addr_len              /* size of this address (16 for internet, variable for UNIX) */
)   /* servers register their well-know addresses with the system, and the system fills in
     * the local_address and local process fields of the struct sockaddr *myaddr */

int connect  ( int sock_desc, struct sockaddr *serv_addr, int addr_len)
    /* results in actual connection between client and server */

int listen  (  int sock_desc,
               int backlog               /* how many connection requests can be queued before
                                          * servers execute an accept() system call */
)   /* indicates the server is willing to receive connections */

int accept  (  int sock_desc,
               struct sockaddr *peer,    /* address of client process */
               int addr_len
)   /* takes the first connection request from the queue of pending requests, creates a socket with the same
     * properties as the server socket. Returns an error or a new socket descriptor */
```