Project 2
Due Wed. 11/23/2004
At the beginning of class
Prof. William A. Maniatty

Abstract
Distributed file access is a common problem in networked environments. In this project you will write a simple library to do networked I/O.

1 Introduction
This document is designed to describe a project where you will create an I/O client, server. To do this you will need to know how to use BSD style sockets interfaces (as covered in class and in our example myserver.c and myclient.c). However, this time, your protocol should be implemented using UDP (datagram sockets) not TCP (streaming sockets). In this particular case, your client side will be implemented as a library, while your server will be implemented as an executable program.

2 Project Overview
This project is a systems programming assignment. For this project, you should implement your library using UDP. In the first pass, you should not worry about error recovery (packet corruption, reordering or loss), and can skip implementing ARQ. Do not discard this version of your software, you will need it to do a performance comparison.

In the second pass you should have the receiver acknowledge packets. You should use stop-and-wait ARQ, and compare the performance between stop-and-wait ARQ and the original model without ARQ.

2.1 Extra Credit
For 25 % extra credit, multiplex the server to handle up to 2 concurrent connections. Your server should support stop-and-wait ARQ.
# 3 Systems Programming Project

You have been asked to write the data communication interface for FRAT (the File Remote Access Tool). Since this is a networking and not a GUI course, it is recommended that you use one window for input and one window for output. It will act similar to an instant messaging system, in that each user of a FRAT server will allow up to `MAX_FRAT_CONNECT` pending connections. Your tool should have the following interface:

```c
// returns connected socket if successful or negative value upon failure
int FRAT_open(const char *hostname, /* Machine running the server */
               const char *filename, /* Name of remote file to open */
               int port, /* Which port is the server using? */
               int *server_errno); /* errno of last server system call*/

// returns 0 upon success, negative value if failure, terminates connection
int FRAT_close(int id, /* the socket to close */
                int *server_errno); /* errno of the last server system call */

// returns number of bytes read or negative value upon failure
int FRAT_read(int fd, /* socket handle */
               void *buf, /* user memory space */
               size_t count, /* number of bytes to read */
               int *server_errno); /* errno of the last server system call */

// returns number of bytes written or negative value upon failure
int FRAT_write(int fd, /* socket handle */
                const void *buf, /* memory location to use */
                size_t count, /* number of bytes written */
                int *server_errno); /* errno of the last server system call */
```

## 3.1 The Client Side

For this assignment, you should implement the `frat` library, which provides the “C” programming language interface as specified in `frat.h`. The file `frat.h` can be found on line by following the source code link in the project FAQ page: http://www.cs.albany.edu/~maniatty/teaching/networks/faqp2.html

**Note:** Your project must interoperate with your own software, we will not test it with others. The test environment will be eve.albany.edu, you can develop anywhere you like, but it must run there.

## 3.2 The Server Side

The server shall establish open data and control ports on a machine, waiting for client connections. The standalone “C” language program should be named “myserver.c”, and the executable generated should have the following command line options:

```bash
myserver port
```
where port is the port number.

Note: You should not use low numbers for your ports (below say 5000 or so) as these ports are reserved for the actual production servers (e.g. ftp and mail servers). Instead (to try to avoid collisions) try computing your control port numbers as follows:

\[ \text{myserver Port} = 10000 + \text{Last 4 digits of Student ID number} \]  

You can use this as a base port number.

### 3.3 A Simple RPC Protocol

You should develop a server program and a client call interface library. The server program is assumed to be running on a host before any client can start, thus you can make a stand alone server program that just runs and accepts client connection requests. Any multitasking of the software needs only be done in the server.

Now let's get on to the business of how the client and server communicate the parameters of the system call. While you are free to implement this a different way, the following course of action outlines an approach I think would work.

The library should provide the a (nearly) transparent interface to the system calls implemented. The mechanism to do this is as follows:

1. The client will issue a call request to the server by writing the following to its socket:

   (a) Write a byte containing one of the values specified in type `MY_RPC_CALL` specifying the type of call being issued to the socket.

   (b) Write all fixed length parameters from left to right order (as they appear in the argument list) to the socket. Be careful to use the htonl conversion on long and short integers.

   (c) Process the list of variable length in left to right order as follows:

      i. Process the list of variable length in left to right order as follows:

         A. Write the long integer, `nbytes`, to the socket indicating the number of bytes in the variable length parameter. Again be careful to use the htonl conversion.

         B. Write the variable length parameter to the socket.

Upon completion, the client should read the results from the socket as follows:

1. The client should read the function return value (if any) and `server_errno` from the socket. Be careful to use the ntoh layer, and preserve the result.

   i. Read the long integer, `nbytes`, from the socket indicating the number of bytes in the variable length parameter. Again be careful to use the htonl conversion.
ii. Read the variable length parameter from the socket.

2. The server will respond to a call request to the server reading the following from its socket:

   (a) Read a byte containing one of the values specified in type `MY_RPC_CALL` specifying the type of call being issued from the socket.
   (b) Read all fixed length parameters from left to right order (as they appear in the argument list) from the socket. Be careful to use the `ntoh` conversion on long and short integers.
   (c) Process the list of variable length in left to right order as follows:
      i. Read the long integer, `nbytes`, to the socket indicating the number of bytes in the variable length parameter. Again be careful to use the `ntohl` conversion.
      ii. If necessary malloc space to hold the variable length data.
      iii. Read the variable length parameter from the socket.

Upon completion, the server should write the results to the socket as follows:

   (a) The server should write the function return value (if any) to the socket. Be careful to use the `hton` layer, and preserve the result.
   (b) The server should write the fixed length results
   (c) The server should write the fixed length results from left to right order to the socket. Again be careful to use the `hton` calls to get the bit/byte ordering correct.
   (d) The client should write the variable length results from left to right as follows:
      i. Write the long integer, `nbytes`, to the socket indicating the number of bytes in the variable length parameter. Again be careful to use the `htonl` conversion.
      ii. Write the variable length parameter from the socket.
      iii. Deallocate any unused space from the variable length parameter if you used malloc earlier.

4 Some Hints

As before you will want to be familiar with:

1. the GNU C compiler (gcc)
2. the GNU debugger, (gdb)
3. the GNU make program
4. the Unix send/sendto/sendmsg system calls

5. the Unix recv/recvfrom/recvmsg system calls

In Unix the documentation can be accessed by the `man` command. Note that system calls are in section 2 of the Unix manual (for historical reasons) and you can access them by doing

```
man 2 systemcall
```

where `systemcall` is the name of the system call you want to invoke. GNU tools are also documented using the `info` facility in emacs.