Washington University
Department of Computer Science

CS422: Operating Systems Organization
Fall 1997

Programming Project
Due October 31^{st}, 1997

1 Overview

The purpose of the following assignment is to acquaint you with using the `socket` system calls that perform network IPC between two processes. The processes may run on the same machine or on different machines. The behavior of this program will be very similar to your previous assignment using pipes, FIFOs, and `fork`/exec. The implementation, however, will be quite different.

2 Development Steps

We will write two programs: (1) a client and (2) a server.

2.1 Concurrent Server

The server program should do the following activities:[1]

1. Create a passive-mode *listener* STREAM socket (`socket(3n), bind(3n), listen(3n)`) and wait for clients to initiate connections (`accept(3n)`). Depending on command-line options given by the user, either create the socket in the Internet domain or in the UNIX domain. Make sure that the port number or the pathname is a command-line option, as follows:
   - The Internet domain address is specified on the command-line via the -i option, *e.g.,* `-i 5001`. Select a port greater than the value `IPPORT_USERRESERVED`. This macro is defined in the `/usr/include/netinet/in.h`.
   - The UNIX domain address is specified on the command-line via the `-u` option, *e.g.,* `-u /tmp/foo`. The argument to `-u` should be a UNIX pathname that is readable and writable by you. I suggest you create a name that won't conflict with other users (*e.g.*, use your login id).
2. When a client request arrives, accept the connection (`accept(3n)`) and spawn a child process to handle the request (`fork(2)`). The child process should dup its standard input to the newly accepted socket descriptor (`dup(2)` or `dup2(2)`) and invoke `/usr/bin/more` to process incoming bytes one screen at a time (`execl`). The parent process should wait for the child

[1] In brackets are some hints about what kinds of library and system calls you might want to look up in order to do these (see the UNIX manual pages and/or the Richard Stevens book on *UNIX Network Programming* for additional details).

process to finish (`wait(2)`), after first closing down the newly accepted socket in the parent (`close`). When the child is finished, the parent should continue waiting for incoming client requests (`accept(3n)`).

Note that the server should not exit unless you explicitly kill it via a signal (`signal(2)`), such as a SIGINT or SIGQUIT sent from the command-line of the machine (`kill(1)`) where the server is running. You should add a signal handler that performs any termination code necessary to gracefully shutdown the server upon receipt of a signal.

2.2 Client

The client program should do the following activities:

1. Read the name of an input file and the addressing information for the remote server process from the command line. Use this information to open the input file for reading (`open(2)`) and to create a socket (`socket(3n)` and `connect(3n)`). The socket should use the same communication domain as the server (*i.e.,* Internet domain vs. UNIX domain). In addition, it should be connected to the server machine using the same addressing information (*i.e.,* hostname/port for Internet domain vs. pathname for UNIX domain). This information is specified on the command-line as follows:

   - The Internet domain address is specified on the command-line via the -i option, *e.g.,* `-i ritz.cec.wustl.edu:5001`. The first part of this address is the hostname of the server, followed by a colon (':') followed by the port number. Select a port greater than the value `IPPORT_USERRESERVED` (defined in `/usr/include/netinet/in.h`. Make sure you use the `gethostbyname(3n)` library function to determine the server host's IP address.
   - The UNIX domain is specified on the command-line via the `-u` option, *e.g.,* `-u /tmp/foo`. The argument to `-u` should be a UNIX pathname that is readable and writable by you.

2. Read all the data from the input file and use the connected socket descriptor to write the data to the remote host (`read(2)`/`write(2)`). Make sure that you handle the case of "short writes."

The client should simply print out the appropriate error message (`perror(3S)`) and exit with a return status of 1 if any of the system calls fail to work properly. If everything works correctly, the program should exit with a return status of 0.

3 Comments

Although this assignment is fairly easy to describe, there are a number of subtleties involved in making it work correctly. In particular, you will need to learn about details such as network addressing, remote host identification, signal handling, process creation and termination, etc. Make sure that you start early and ask lots of questions.

In addition, please note that although we will use the `socket` network IPC services, we may actually communicate between processes on the same host. This is useful for testing purposes. However, the same programs will work for communicating between processes on two separate host machines! For example, if you use the Internet domain, you can set up the server on `clarion` and run the client on `ritz` or vice versa.

4 Extra Credit

4.1 C++ Socket Wrappers

Once you get the basic program working you should consider encapsulating the socket interfaces with C++ wrappers. The goal is to make it easier and safer to program with sockets. The following are some design principles that you should use when creating your socket wrappers:

- Enforce typesafety at compile-time
- Allow controlled violations of typesafety
- Simplify for the common case
- Replace one-dimensional interfaces with hierarchical class categories
- Enhance portability with parameterized types
- Inline performance critical methods
- Define auxiliary classes to hide error-prone details

You might want to take a look at the paper at URL
`www.cs.wustl.edu/~schmidt/IPC_SAP-92.ps.gz`
for more details on C++ socket wrappers.

4.2 Dyanmic Selection of C++ Socket Wrappers

If you use ACE, you can get extra credit by developing a set of C++ adapter classes for the ACE socket wrappers. These adapter classes will make it possible to select whether you use Internet domain sockets or UNIX domain sockets *at run-time*.

Here's what you could do:

```
class IPC_Stream
{
public:
  virtual ssize_t send (const void *buf, size_t n,
                        int flags) const = 0;
  // Send an n byte buffer.

  virtual ssize_t recv (void *buf, size_t n,
                        int flags) const = 0;
  // Recv an n byte buffer.

  // ... etc.
};
```

and then you could create a template adapter, as follows:

```
template <class STREAM>
class IPC_Stream_Adapter : public IPC_Stream
{
public:
  IPC_Stream_Adapter (const STREAM &s)
    : stream_ (s) {}

  virtual ssize_t send (const void *buf, size_t n,
                        int flags) const
  {
    return this->stream_.send (buf, n, flags);
  }

  virtual ssize_t recv (void *buf, size_t n,
                        int flags) const = 0;
  {
    return this->stream_.recv (buf, n, flags);
  }

  // ... etc.

private:
  STREAM stream_;
  // IPC mechanism (e.g., ACE_SOCK_Stream, ACE_LSOCK_Stream).
};
```

Once you've got this, you can create the appropriate specializations, as follows:

```
IPC_Stream *
stream_factory (void)
{
  IPC_Stream *stream;

  if (use_inet_sock)
    stream = new IPC_Stream_Adapter<ACE_SOCK_Stream>;
  else
    stream = new IPC_Stream_Adapter<ACE_LSOCK_Stream>;

  return stream
}
```

The details of doing all this correctly are left to the implementor.