Project 1: Implement a simple hosts framework using UNIX TCP Sockets to demonstrate the concept of Mobile Agents

Due date: October 15 (Wednesday), 2003

Requirements

This exercise should be done in a group of at most two students. You will be asked to demonstrate your programs and to submit well commented codes and report via soft (floppy) and hard copies. In your codes and report, the contribution of each group member needs to be specified.

Exercise 1: A server that echoes client input

You are required to produce client and server programs based on the procedures DoOperation, GetRequest and SendReply. These operations have been simplified so that client and server exchange messages consisting of strings.

The interaction between the client and server should be as described below:

Client: this takes the name of the server computer as an argument. It repeatedly requests a string to be entered by the user, and uses DoOperation to send the string to the server, awaiting a reply. Each reply should be printed out.

Server: repeatedly receives a string using GetRequest, prints it on the screen and replies with SendReply. The server exits when the string consists of the single character ‘q’.

Use the following (or equivalent C) definitions for Status and SocketAddress, which are in C++.

```c
enum Status
{
    OK, // operation successful
    BAD, // unrecoverable error
    WRONGLENGTH // bad message length supplied
};
```

typedef sockaddr_in SocketAddress;

Implement DoOperation, GetRequest and SendReply methods. The recommended prototypes are given in the definitions in the appendices. In the C++ definitions they are included in the classes Client and Server. The Status value returned reflects the values returned by TCPsend and TCPreceive (see below).

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoOperation</td>
<td>Sends a given request message to a given socket address and blocks until it returns with a reply message.</td>
</tr>
<tr>
<td>GetRequest</td>
<td>Receives a request message on a connected socket.</td>
</tr>
<tr>
<td>SendReply</td>
<td>Sends a reply message to the host through a connected socket.</td>
</tr>
</tbody>
</table>
**TCPsend and TCPreceive**

The procedures DoOperation, GetRequest and SendReply must use two procedures TCPsend and TCPreceive to be written by you, which respectively send and receive a message over/from a socket. You are to implement these functions using the system calls sendto/send and recvfrom/recv respectively.

Each procedure returns a value of type Status which reports on the success of its execution. For example, if the sendto/send or recvfrom/recv system calls return negative values, your procedures should return a Status value of Bad.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCPsend</td>
<td>Sends a given message through a connected socket.</td>
</tr>
<tr>
<td>TCPreceive</td>
<td>Receives a message on a connected socket.</td>
</tr>
</tbody>
</table>

Use the recommended definitions for SocketAddress and Message and the recommended prototypes for TCPsend and TCPreceive. In the C++ definitions, the latter are to be found in class Socket.

**Choosing a server port**

You will want to run server processes that can coexist with other people's processes in the same computer. You need to select an agreed port number for the server to receive messages from clients. Two servers on the same computer cannot use the same local port number. You will therefore want to choose a port number that is sure to be different from other people's port numbers. If everybody takes the first unreserved port number and adds their uid, there should be no such clashes - i.e.:

\[
\text{aPort} = \text{IPPORT_RESERVED} + \text{getuid}() ;
\]

**How to write an Concurrent Server?**

As part of this exercise, you should implement the TCP echo server as a concurrent server and try to demonstrate the concept by using the same server to server more than one client at the same time. *Describe the experiment and discuss its results in a comment in your report.*

**Hint**: Use pthreads API to implement concurrent a server.

**Exercise 2: Implement a Mobile agent on a set of hosts connected through TCP/IP sockets.**

In this exercise you will need to extend the concepts from the Exercise 1, wherein you have established a connection between the client and the server. Here, you need to implement three hosts each of which is a client and server.

**Mobile Agents:**

An agent is an independent software program that runs on behalf of a network user. A mobile agent is a program, once launched by a user, can travel from host to host.
autonomously and can contribute to the functionality even if the user is disconnected from the network. But, we would be implementing it for a set of three connected hosts.

Figure 1: A mobile agent travels from host to host.

An agent carries following data items with it:
1. Identifier – information that allows an agent to be identified.
2. Itinerary – a list of addresses/names of the hosts that the agent is to visit.
3. Task data – data required by the agent to perform its tasks.
4. Process logic – logic(code) to perform its tasks on the task data.

In this exercise you will create an agent object which consists of all of the above described data. This agent object will be launched by a user, say host 1 in the figure 1. On reaching host 1 it should execute it logic using the resources of the host server. At each host server, it needs to be examined, if there are any more hosts that need to be served by the agent before returning home. If so, then the current host server will update the next host index to be served by the agent and transport the agent to the next host server. Once the mobile agent exhausts all the hosts to be served it is assumed that it has reached home and thus completed its cycle.
<table>
<thead>
<tr>
<th><strong>STOP</strong></th>
<th>The host server returns <em>STOP</em> to the initiating user/host and the cycle stops.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CONTINUE</strong></td>
<td>The host server returns <em>CONTINUE</em> to the initiating user/host and cycle continues.</td>
</tr>
</tbody>
</table>

Client and server should provide the ability to use *STOP* and *CONTINUE* to stop, and continue services of the mobile hosts respectively.

**Initiator:** This is network user/host is responsible for launching the mobile agent. It marshals the data of the agent into a Message object and transports it using TCP sockets to the next host server in the itinerary of the agent.

**Server Hosts:** Each of these hosts receives the mobile agent in form of a Message object, unmarshal the data contained in the message object to reconstitute the agent, at which the process logic embedded in the agent it executed using the host’s resources. The host then looks up the if there are any hosts remaining to be serviced by the agent and re-marshals the agent into a Message object to transport it the next host.

This exercise requires you to write marshalling and unmarshalling function members, which take network ordering into account, by using the functions *htonl* and *ntohl*. The recommended prototypes for marshal and unmarshal are given in the definitions. In C++ they are function members of the class *MAgent* (described below in Appendix 1).

Two other matters that should be addressed:

- The mobile agents need to have an identifier which is known only to the participating hosts and the agent. This is important from the security perspective.
- The hosts need to output the data modified by applying the process logic stored in the agents. The process logic can be unary or binary arithmetic operators, which are enumerated and have a predefined meaning so that the hosts can perform the corresponding operations on reading the enumerations. You can perform operations like addition, subtraction, multiplication, division, square root, square, etc.

**Demonstrations**

**Exercise 1**

You must demonstrate that your programs work correctly by running two clients and a server on three different computers. Also, demonstrate the concept of concurrent servers by allowing more than one clients being served by a single server at the same time.

**Exercise 2**
One of the three host servers should be an initiator which launches the mobile agent. You should demonstrate this by printing some text statements. On reaching the host the mobile agent’s data should be operated on based on the logic stored in it and the corresponding result needs to be demonstrated on the server. Same holds for the second host the mobile agent visits. Once the mobile agent reaches home and its state is reconstituted, the resulting data in the agent should be printed out by the receiving host which was the initiator at the start of the itinerary of the agent.

**Print-outs to hand in**

The client and server programs should have separate codes. Also, the mobile agent should have

**Exercise 1**

Provide the main program for the client and the server. The code for the client should include a comment containing a short write-up describing the implementation of the concurrent TCP server.

**Exercise 2**

Please supply the print outs in the following order:

- Header files with definitions for `SocketAddress`, `Status`, `Message`, `RPCMessage`. C++ programs will provide class interfaces, C programs will provide function prototypes;
- Implementation files for the classes or functions;
- Separate header and implementations for the Arithmetic service;
- A main for the client and the server (only one of each).

---

**APPENDIX 1: Definitions for C++ Programs**

In these exercises you are advised to define the following:

---

class Message {
public:
    Message(unsigned char *, unsigned int ); // message and length supplied
    Message(unsigned int ); // length supplied

private:
    unsigned char * data;
    unsigned int length;
};

The class `Message` is for holding the data and the length of a message. If you prefer you can make data contained in the Message class a fixed length array of size 1000. The first
constructor is used to create a message with given contents. The second constructor is used to create a message with a given length for putting the data into later. You may find that you need other function members, e.g. to return the data or length.

class Socket {
public:
    Socket();
    Socket(int); // port number passed as an argument
    status TCPsend(TCPMessage *m, SocketAddress * destination);
    status TCPsend(TCPMessage *m, int sockfd); // sockfd is the connected socket
    status TCPreceive(TCPMessage **m, SocketAddress * origin);
    status TCPreceive(TCPMessage **m, int sockfd); // sockfd is the connected socket
private:
    int s;
    SocketAddress * socketAddress;
};

The class Socket represents a stream socket. It has data members for the socket descriptor and for the socket address to which it is bound. There are two constructors i) no arguments: opens the socket and binds it to any local port, ii) one argument: opens a socket and binds it to the given port. The function members send and receive messages via the given port.

class Client: public Socket {
public:
    Client(); // calls constructor Socket()
    status DoOperation (TCPMessage *callMessage, TCPMessage *replyMessage,
                        SocketAddress * server);
};

class Server: public Socket {
public:
    Server(int); // calls constructor Socket(int);
    status GetRequest (TCPMessage *callMessage, SocketAddress * client);
    status GetRequest (TCPMessage *callMessage, int connfd);
    status SendReply (TCPMessage *replyMessage, SocketAddress * client);
    status SendReply (TCPMessage *replyMessage, int connfd);
};
You can define another class or classes whose instances have the operations *DoOperation*, *GetRequest* and *SendReply*. One way to do this is to define two new classes called *Client* and *Server*, both of which are subclasses of the *Socket* class, as shown above.

**C++ definitions for Exercise 2:**

Declaration and description for an example Mobile agent class:

```cpp
typedef enum { ADDITION,
               SUBTRACTION,
               MULTIPLICATION,
               DIVISION,
               SQUARE
} OP_TYPE;

enum MessageType { Request, Reply};

class MAgent {
    void MAgent (unsigned int, OP_TYPE , char **, int , double a1 = 0, double a2 = 0);
    void marshal( Message ** );     // marshal: encodes itself to a Message object
    void unmarshal(Message *);   // unmarshal: decodes from a given Message
                                //object to itself
    void execute ();   // executes the operation on the task data

protected:
    unsigned int m_Identifier;  // agent identifier
    int m_hostIndex;             // which host to visit next
    char **m_host_list;             // list of hosts
    OP_TYPE m_operationID;  // stores the enumerated operation type
    double m_arg1, m_arg2;            // arguments for the operations (task data)
};
```

The list of hosts to be visited by the agent in its itinerary as contained in the variable *m_host_list*, and *m_hostIndex* holds the value of the next host to be visited. Both of these variables are looked up, and based on their current values and the next host to be visited is decided by the currently visited host/server. The variables *m_arg1* and *m_arg2* is the task data contained in the agent, which are operated on, based on the enumerated type (OP_TYPE) stored in the variable *m_operationID*. *MAgent* class has a constructor which takes the agent identifier, enumerated operation type, list of hosts to be served, next host index, and the task data as the parameters. The
two functions *marshal* and *unmarshal* flatten and reconstitute the execution state of the agent object respectively. *Execute* function is used to execute the process logic stored in the agent on the data stored in the agent. The process logic can be any of the operations defined in the type OP_TYPE; you can also add more operations to the enumerated type shown above.

**APPENDIX 2: Definitions for C Programs**

In these exercises you are to use the following type definitions:

```c
#define SIZE 1000
typedef struct {
    unsigned int length;
    unsigned char data[SIZE];
} Message;

typedef enum {OK,   /* operation successful */
              BAD,   /* unrecoverable error */
              WRONGLENGTH /* bad message length supplied */
} Status;

typedef struct sockaddr_in SocketAddress;
```

You may alternatively define *data* as a pointer.

The prototypes for *DoOperation*, *GetRequest* and *SendReply* (to which you must adhere) are as follows:

```c
Status DoOperation (Message *message, Message *reply, int socket_descriptor,
                    SocketAddress *serverSA);

Status GetRequest (Message *callMessage, int connfd, SocketAddress *clientSA=NULL);

Status SendReply (Message *replyMessage, int connfd, SocketAddress clientSA=NULL);
```

The prototypes for *TCPsend* and *TCPreceive* are as follows:

```c
Status TCPsend(int connfd, Message *m, SocketAddress destination=NULL);
Status TCPreceive(int connfd, Message *m, SocketAddress *origin=NULL);
```

**C definitions for Exercise 2**
You should use the following definition of an agent:

typedef enum {  ADDITION,
                 SUBTRACTION,
                 MULTIPLICATION,
                 DIVISION,
                 SQUARE
             } OP_TYPE;

typedef struct {  
     unsigned int m_Identifier;  /* unique identifier */
     int m_hostIndex;              /* which host to visit next*/
     char **m_host_list;           /* list of hosts*/
     OP_TYPE m_operationID;        /* e.g.(0,1,2,3) for (+, -, *, /) */
     int m_arg1;   /* argument/ return parameter */
     int m_arg2;   /* argument/ return parameter */
} MAgent;    /* each int (and unsigned int) is 32 bits = 4 bytes */

The fields arg1 and arg2 can be used for operation arguments or returned value and status.
The prototypes of the marshalling and unmarshalling procedures should be as follows:

    void marshal(MAgent *rm, Message *message);
    void unMarshal(MAgent *rm, Message *message);