CS240 Operating Systems, Communications and Concurrency – Dermot Kelly
Practical 6 - Unix Interprocess Communication Mechanisms
Socket Communication and Remote Method Invocation

In the third (and last) of our practicals on Interprocess Communication Mechanisms, we take a look at Network Communication Mechanisms for communication between unrelated processes running on different machines. First we look at low level Socket based communication in C. Then we show how Java can simplify socket programming with predefined networking classes and specific constructors with few parameters. Finally we present the higher level programming model of Remote Procedure Call/Remote Method Invocation which attempts to mask as much of the underlying network connection code as possible. Take your time to read this handout before compiling any code.

Sockets

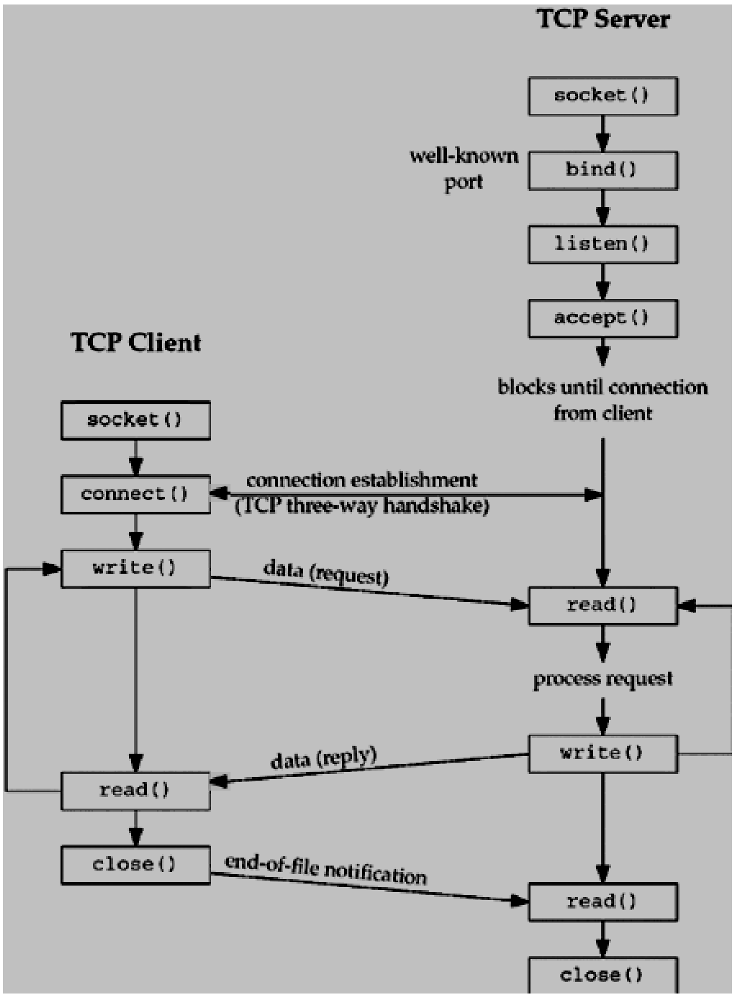
Unix pipes were introduced as a simple means of communication between two processes that doesn't require more complex software networking protocols. However, the pipe mechanism is limited in a number of ways to be useful for general communication. With ordinary pipes, implementation is based on a memory based circular buffer and the communicating processes must be related to each other in order to share access to the pipe descriptors. Unix ***named*** pipes can overcome this for unrelated processes but instead require a common file system between the communicating processes that can be used to name and implement the pipe. A pipe is used as a one-way data stream connection between two processes. To establish two-way communication usually requires the creation of two pipes. The class of service always implements reliable, FIFO ordering between the communicating processes, that is, the stream of bytes written to the pipe is read in the same order at the other end.

What mechanism can be used when the communicating processes are not related and do not share a file system? This requires a common transport level communication subsystem to be resident on each host involved in the communication such as TCP/IP. TCP/IP is the common message transport protocol suite used on the Internet. This subsystem routes and delivers messages from one endpoint on a network to another. The Unix socket interface is a way of binding a reference object in a program's address space (a socket) with a communication endpoint (a port) in the underlying message transport subsystem to enable message transfer to take place between the endpoints connected by network software.

Think of it in the same way as you did for programs that use files. To use a file you use the `open()` system call to create a binding between your program and the file object stored within the file system. The `open()` call returns a file descriptor, a number by which you identify the file in your program and use in subsequent system call operations on that file. When done with the file you close() it and the system deallocates any memory resources used for mapping your logical file descriptor onto an actual file in the file system.

With network communication, a server first creates a local socket and binds it to a unique port address in the communication subsystem. It then tells the communication to make this a listening port so that it can create an associated queue of incoming requests. The server then waits for incoming connections. A client creates a local socket and asks the communication subsystem to connect its socket to the remote server using the server's well known port number. Each accepted connection establishes a new temporary server side socket, unique for handling that particular client/server exchange. This allows multiple threads for example to be handling separate connections from clients using the same server port number.

After the connection is accepted and established, communication takes place by writing into the client's local socket, the data is transparently carried by TCP/IP through the network and may be received through the server's socket at the other end. The server parses and acts on the request and then may write a response into its own socket and close the connection. The response is then carried back to the client over TCP/IP, the client will read the response from its socket and may then close the connection on its side.



The basic sequence of calls to the socket interface for implementing client server message exchanges is given on the previous page.

The code that follows demonstrates the use of sockets to enable communication between a client program that sends a greeting message in plain text to a server, and a server program that receives messages from arbitrary client processes and sends back text responses. This type of exchange is the basis of the HTTP protocol used by the World Wide Web (WWW).

Study the code and ensure you understand it generally, it is well commented and you can use the man utility to read further about any of the system calls used. Compile and run the client and server code and make sure you have message exchange taking place between the two independent processes. Once the server is running, you should be able to run the client multiple times. You can use the kill system call to terminate the server at any time and release its port.

See if you can modify the code (change the IP address) so that your client talks to your neighbour's server rather than the one running on your own machine.

The code is minimal to help explain the steps of socket communication more clearly but as a consequence it does not print any error messages if things go wrong with various system calls or if ports are in use. You can add your own error messages. Such error handling code would be part of any proper implementation. **As an exercise, add a piece of error checking code to check that the server created a socket successfully at the start of the code and terminate with an error message if not. Now run this code at least twice. The error message should be displayed when you try to run the server code a second time because the port it wants to open is in use.**

Save the program below as `socketclient.c` and
compile with `cc socketclient.c   -o socketclient`

```
/**************** Client Code ***************/
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_addr() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define RCVBUFSIZE 32   /* Size of receive buffer */

int main(int argc, char *argv[])
{
    int sock;                         /* Socket descriptor */
    struct sockaddr_in echoServAddr; /* Echo server address */
    unsigned short echoServPort =8093;      /* Echo server port */
    char *servIP;                     /* Server IP address (dotted quad) */
    char *echoString;                 /* String to send to echo server */
    char echoBuffer[RCVBUFSIZE];      /* Buffer for echo string */
    unsigned int echoStringLen;       /* Length of string to echo */
    int bytesRcvd, totalBytesRcvd;    /* Bytes read in single recv()
                                         and total bytes read */
    servIP = "127.0.0.1";  /* server IP address (this host's own address) */
    echoString = "hello";   /* string to echo */
    echoStringLen = 5;
    echoServPort = 8093;      /* port for the echo service */

    /* Create a local client stream socket using TCP */
    sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

    /* Construct the server address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));     /* Zero out structure */
    echoServAddr.sin_family      = AF_INET;             /* Internet address family */
    echoServAddr.sin_addr.s_addr = inet_addr(servIP);   /* Server IP address */
    echoServAddr.sin_port        = htons(echoServPort); /* Server port */

    /* Establish the connection to the server */
    connect(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr));

    /* send the message over the socket connection */
    send(sock, echoString, echoStringLen, 0);
    printf("Client: Sent greeting-> %s\n", echoString);

    /* Receive a reply back from the server */

    /* Receive up to the buffer size (minus 1 to leave space for
           a null terminator) bytes from the sender */
    bytesRcvd = recv(sock, echoBuffer, RCVBUFSIZE - 1, 0);

    echoBuffer[bytesRcvd] = '\0';  /* Terminate the string! */
    printf("Client: Received Reply-> %s\n", echoBuffer); /* Print the string received
from server */

    close(sock);
    exit(0);
}
```

Save the program below as `socketserver.c` and
compile with `cc socketserver.c  -o socketserver`

```
/*************** Server Code ***************/
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), bind(), and connect() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define MAXPENDING 5    /* Maximum outstanding connection requests */
#define RCVBUFSIZE 32   /* Size of receive buffer */

int main(int argc, char *argv[])
{
    int servSock;                    /* Socket descriptor for server */
    int clntSock;                    /* Socket descriptor for client */
    struct sockaddr_in echoServAddr; /* Local address */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned short echoServPort;     /* Server port */
    unsigned int clntLen;            /* Length of client address data structure */
    char echoBuffer[RCVBUFSIZE];     /* Buffer for receiving client's msg string */
    int recvMsgSize;                 /* Size of received message */
    char *echoString;                /* Server's reply to client */
    unsigned int echoStringLen;      /* Length of server's reply string */

    echoString = "server is alive, how are you?";   /* Server's reply to client */
    echoStringLen = 29;
    echoServPort = 8093;  /* local port on which server is going to listen */

    /* Create local TCP/IP socket for incoming connections */
    servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

    /* Construct local address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));   /* Zero out structure */
    echoServAddr.sin_family = AF_INET;                /* Internet address family */
    echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
    echoServAddr.sin_port = htons(echoServPort);      /* Local port */

    /* Bind local socket to the desired server port address */
    bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr));

    /* Mark the socket so it will listen for incoming connections */
    listen(servSock, MAXPENDING);

    for (;;) /* Run forever */
    {
        /* Set the size of the in-out parameter */
        clntLen = sizeof(echoClntAddr);

        /* Blocking wait for a client to connect */
        clntSock = accept(servSock, (struct sockaddr *) &echoClntAddr,&clntLen);
        /* clntSock is connected to a client! */
        printf("Server: Handling client %s\n",inet_ntoa(echoClntAddr.sin_addr));

        /* Receive message from client */
        recvMsgSize = recv(clntSock, echoBuffer, RCVBUFSIZE, 0);
        printf("Server: Received msg-> %s\n",echoBuffer);

        /* Send response message back to client */
        send(clntSock, echoString, echoStringLen, 0);
        printf("Server: Sent Reply-> %s\n",echoString);
        close(clntSock);    /* Close client socket */
    }
}
```

Using Socket Communication in Java Programs

Java bytecode programs run in java virtual machines. A virtual machine is an execution environment with a specific instruction set which maps to the instruction set of the actual processor it runs on. There can be many different implementations of the Java virtual machine for different processor architectures and operating systems. As long as there is an implementation of the Java virtual machine for a particular platform, you can run any Java program on it. This makes Java programs portable across many types of architecture and explains why Java has become popular.

Programs running on Java virtual machines can communicate using Java network services mapped on top of the interprocess communication subsystems of the host operating environments. As Java is an object oriented language, I/O operations sometimes are difficult to explain which is why we used the C programming language in earlier practicals along with system calls like `open()`, `close()`, `read()` and `write()` to get a better feel of what was happening with input/output streams at a lower level without being confused by objects types and methods. Next we are going to show two ways in which networked communication can be done in Java programs.

The first is to use Java socket based streamed communication where a server creates a local socket, binds it to a communication port and accepts connections from clients who know its address. This is the same concept used earlier in the practical except that the Unix socket API was used with the C programming language.

When the sockets are connected, data can be read and written using `InputStreamReader` and `PrintWriter` objects.

In the example below, a client process connects to a server process to obtain the date on the machine on which the server is executing.

```
// This is the Server code, save as DateServer.java
import java.net.*;
import java.io.*;

public class DateServer {
   public static void main(String[] args) throws
                                          IOException {
      try {
         // This creates a listener socket
         ServerSocket sock = new ServerSocket(6013);
         while (true) {
            Socket client = sock.accept();
            // pout is the output stream to the client
            PrintWriter pout = new
                 PrintWriter(client.getOutputStream(),true);

            pout.println(new java.util.Date().toString());

            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

```java
// This is the Client Code, save as DateClient.java
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args) throws IOException {
      try {
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         // bin is the input stream from the server
         BufferedReader bin = new
               BufferedReader(new InputStreamReader(in));

         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Compile the programs

```
javac DateServer.java
javac DateClient.java
```

Start a separate terminal window and **Run the programs**

```
java DateServer &
java DateClient
```

Note that this type of communication can work across different physical machines by using the appropriate IP and port addresses in the client as you did earlier.

Note how a lot of the low level functionality in the C code example is masked from the programmer in the Java version by using the functionality of the Socket and ServerSocket classes whose constructors provide simpler interfaces with fewer parameters for establishing socket connections and allow for easier expression of formatted reading and writing of the stream. Although more complex, the lower level C code can be more flexible in some situations because using a preconfigured constructor in Java means we have to use all the default behaviour of the constructor.

Remote Method Invocation in Java

The second (higher level) programming option for communication is to use Java Remote Method Invocation (RMI). With this scheme, an abstract interface to a service is declared. The server program creates an actual object instance to implement the service methods defined in the interface and this object is then bound to a text name for the service registered with a persistent `rmiregistry` process. A client can discover and connect to a service (for which it implements the abstract interface), by consulting the `rmiregistry` process, like a telephone book. The `rmiregistry` returns the object reference for the remote object which can be used with the help of an interface stub class to invoke methods as if the object were local.

Note the higher level of abstraction for the programmer in the RMI code. The handling of connections an communication with the server is implicit and automatic, and the way the client connects and passes parameters is much cleaner than the socket version we did earlier.

Further details on the latest Java API may be found on the official Oracle website at
http://docs.oracle.com/javase/8/docs/api/index.html

Remote Method Invocation

An interface must be declared that specifies the methods that can be invoked remotely. This interface must extend the `java.rmi.Remote` interface as shown below. Each method declared in this interface must throw `Remote` exceptions. The server will implement one remote operation known as `getDate().`

```
// This is the interface definition, save as RemoteDate.java
import java.rmi.*;
import java.util.Date;

public interface RemoteDate extends Remote
{
   public abstract Date getDate() throws RemoteException;
}


// This is the server code, save as RemoteDateImpl.java
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class RemoteDateImpl extends UnicastRemoteObject implements RemoteDate
{
   public RemoteDateImpl()throws RemoteException {}

   // This is the implementation of the remote method getDate()
   public Date getDate() throws RemoteException {
      return new Date();
   }

   public static void main(String[] args) {
      try {
         RemoteDate dateServer = new RemoteDateImpl();
         Naming.rebind("DateServer", dateServer);
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

```
// This is the RMI client, save as RMIClient.java
import java.rmi.*;

public class RMIClient
{
   public static void main (String args[]) {
      try {
        // 127.0.0.1 is IP address of local host
        String host = "rmi://127.0.0.1/DateServer";

        // lookup the rmiregistry and get an object reference for the server
        RemoteDate dateServer = (RemoteDate)Naming.lookup(host);

         // Here is our remote method invocation
        // In one line we are connecting (transparently) with the
        // server object and printing the response received.
        System.out.println(dateServer.getDate());
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

NB: Remote methods and Naming.lookup can throw exceptions.

Running the programs

1. Compile all three source files using `javac`

Step 2. (Optional - Deprecated) Generate the client communication stub using the RMI interface compiler on `RemoteDateImpl`.

```
rmic RemoteDateImpl
```

A stub is a proxy for a remote method, it resides with the client. When a client invokes a remote method, the stub for the remote method is called which parcels the method name and parameters. These are passed to the server where the skeleton for the remote object unmarshalls the parameters and invokes the desired method. This step is unnecessary as the stub can be dynamically fetched at run time.

3. Start the registry, this is where any server will register its name and port to allow connections.

```
rmiregistry &
```

4. Run the remote server object in a JVM

```
java RemoteDateImpl &
```

This remote object will register using the name `DateServer`.

5. Access the remote server object from a client program running in a separate JVM. The stub class generated earlier must be in the same directory location as `RMIClient`.

```
java RMIClient
```

Look at the code. In the two Java implementations, did you see the greater simplicity with RMI compared to socket based communication? Remember though, with simplicity comes lack of flexibility to alter the way the protocol behaves.