Understanding the programming techniques for client-server architectures

V. Dobrota,[a] S.D. Bate,[b] M. Cosma,[a] D. Zinca[a]
[a]*Technical University of Cluj-Napoca, Department of Communications, 3400 Cluj-Napoca, Romania*
[b]*Coventry University, School of Engineering, Coventry, CV1 5FB, UK*

Abstract

This paper sets out the authors' experience in teaching programming techniques applied for the Client-Server architectures. First, as there are multiple requirements for students coming from different areas of computer science or of communications engineering it was decided to find the suitable methods to teach the subject. Second, two approaches have been studied, from the standpoint of: as a future software engineer working in setting up a telecommunications application and as a future communications engineer working in setting up a software tool. The variety of opinions is confirmed by the students' solutions to a specific problem which has been given for a rapid implementation. C was chosen for DOS/Windows and Erlang for UNIX, internetworking with TCP/IP.

1 Introduction

The importance of the client-server architecture is increasing constantly, as more and more people become involved in global communication. The Internet and the Information Superhighway are exciting subjects that are of great interest. It is becoming very clear that in a short time a client-server literacy will be an obligatory condition to gain employment for certain IT personnel. This paper focuses on understanding the mechanism of a client-server architecture in general and some of the particular programming techniques to implement it. Being a relatively new topic to teach an experiment project has been launched for students studying their M.Sc. in telecommunications. Two perspectives have been chosen: 1. *As a future software engineer working in*

Transactions on Information and Communications Technologies vol 10, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

setting up a telecommunications application based on client-server architectures (mainly focused on computer engineering); 2. *As a future telecommunications engineer working in setting up a software tool* based on client-server architectures (mainly focused on communications engineering).

Some teaching problems and difficulties which were overcome during this experiment are also presented. These are accompanied by relevant commentary and practical examples, written in C for DOS/Windows and Erlang for UNIX.

2 Client-server architectures

First it was useful to define what is the meaning of client or server software and what is their specific interworking mechanism. A client is an application which initiates peer-to-peer communication. Most client software consists of conventional programs which contact the server, send a request and wait for an answer. If the answer comes in time, the client continues its process. There are some standard applications included in this category and mentioned by Stallings [2], such as TELNET, FTP, SNMP (defined by TCP/IP). Some examples of non-standard applications (at the moment) could be an experimental audio-video conferencing system, specific access to a distributed database etc.

A server is an application that expects communication requests from its clients in order to provide the service and to return back the result towards the originator. TCP/IP will be considered in our case study. The technology frequently used in this area includes CO *(Connection-Oriented)* servers (like TCP) and CL *(Connectionless)* servers (like UDP).The TCP servers are suitable for a reliable communication channel and Comer & Stevens [1] strongly recommended them to be implemented by the beginners. There is a permanent check if data has been received error-free at its destination, a mechanism of retransmission being activated if necessary. The UDP servers do not offer guaranteed data delivery so there is no confirmation, like in the previous case. They are suitable for broadcasting applications or whenever the protocol overhead is too long to be acceptable (Berson [3]).

Obviously the attribute of being a client or a server is relative: in the mean time an application could request a communication from a server (acting as a client) and also could provide itself services for other clients (acting as a server). This duality has an important influence over the programming techniques used to implement the desired architecture.

3 Problems in teaching and learning of internetworking TCP/IP - TLI Version

AT&T has launched TLI *(Transport Layer Interface)* as an interface between application program and the protocol software in the operating system, being in fact a transport service supplier. The specifications, that have been published by Comer & Stevens [1], are applied to the following defined functions:

Transactions on Information and Communications Technologies vol 10, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517


```
t_accept     t_free       t_optmgmt      t_rcvuderr
t_alloc      t_getinfo    t_rcv          t_snd
t_bind       t_getstate   t_rcvconnect   t_snddis
t_close      t_listen     t_rcvdis       t_sndrel
t_connect    t_look       t_rcvrel       t_sndudata
t_error      t_open       t_rcvudata     t_sync
                                         t_unbind
```

The students' reaction to this new subject was dependent on their background (whether in computer or communications engineering) and their programming skills. It was supposed that they had already learnt to write programs in C and Intel 80x86 assembly language. In addition some of them became familiar with a special programming language (Erlang), which is suitable for telecommunications applications, as in [5]. The client-server architecture is a topic covered in a general course: 'Computer Networks'.

```
t_open(path,oflags,info);

DESCRIPTION: Both clients and servers call t_open to
create a communication descriptor.The caller specifies
a path associated with the transport service provider
and the type of service.
ARGUMENTS:
path &char               A pointer to a path name for
                         transport provider
oflags int               The same as the flag bits in
                         UNIX's open
info &struct t_info      Other information including
                         the service type
RETURN CODE: t_open returns a nonnegative communication
descriptor if successful and -1 if an error occurs.
When an error occurs, the global variable t_errno
contains: TBADFLAG, TSYSERR
```

Figure 1: Example of a TLI function, Comer & Stevens [1]

The most sensitive terms (from the teaching point of view) were the following: definition of client, server, port, communication descriptor, global variable. After the presentation of the specification for TLI functions were completed, almost 60% of students were very confused about an apparent lack of information provided. The implementation seemed to be very difficult at the beginning, mainly because the simplicity of the mechanism had not been understood. When a programmer writes an application which must be integrated later in a client-server architecture, he does not need to know what the operating system has to do in order to open, bind, connect, send, receive or close communication. Because of the TLI functions, he just needs to know how to address them, irrespective of whatever operating system is currently running.

Therefore the teaching experiment included a one-term project by organizing the students in four teams: client TCP, client UDP, server TCP, server UDP. Each member of a team receives a TLI function to be implemented and then

Transactions on Information and Communications Technologies vol 10, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

after the integration of their work in a client or a server software, a peer-to-peer experiment is performed. A software package *PC/TCP for DOS/ Windows* (FTP Software) is assumed to be installed, with a network interface (could be packet drivers/NDIS/ODI). All the services provided by *int 61h* become available after the executing of *ethdrv.exe* , which remains resident. The next paragraph suggests some professional problems that could be considered.

4 Programming techniques

4.1 DOS/Windows version of a client-server architecture

Conceptually, TLI functions for Windows applications might be implemented in two ways. In the first one a shared library, called DLL *(Dynamic Linked Library)* in Windows, is used. This allows the system to keep a single copy for every function. The second way to implement TLI functions assumes a conventional library to be statically linked to every program if necessary. Therefore, every program that uses them will include its own copy of the needed function linked to it, suitable for DOS implementation.

4.1.1 TLI functions, data structures and symbolic constants What is essential for the TLI interface is to define a new type of data called *com_descriptor*. It is analogous to the DOS files descriptors. This small integer is used as an index into a table to identify the communication path to a particular peer. The declaration will be contained in the *tli.h* header file: `typedef short com_descriptor;` The next step is to define all the data structures that will be needed. For example, the function *t_open* needs a data structure, called *t_info,* that will contain information about the communication requested to be set. The declaration for this structure will be also in the *tli.h* header file (Figure 2). Then, by including the header file *tli.h*, every source program will be able to "know" about this data structure. All the six other data structures that are needed by the other TLI functions (*t_bind, t_call, t_discon, t_optmgmt* etc.) will be defined in the header file in the same way as *t_info*.

```
typedef struct
     {
     long addr;          //size of protocol  address
     long options;       //size of protocol options
     long tsdu;          //size of max service data unit
     long etsdu;         //size of max expedited TSDU
     long connect;       //max data sent during connection
     long discon;        //max data sent during disconnection
     long servtype;      //transport provider service type
     } t_info ;
```

Figure 2: Example of a data structure used for a TLI implementation

The header file will also include the definitions of all the TLI functions implemented. Thereafter, all these functions can be used by the programmers, instead of learning about the operating system details and writing the own TLI

Transactions on Information and Communications Technologies vol 10, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

functions. Then it is the job of the compiler (if a conventional library is to be implemented) or the Windows system (if a DLL is to be implemented) to link the copies of the TLI functions to all programs that will request to use them. In addition to data structures and functions, the interface must provide a set of predefined symbolic constants. These constants may be used in applications to specify arguments, global variables or return values. They have to be defined in the header file in statements like `#define TSYSRERROR 8` . To use all these predefined constants, structures and functions into C applications, an *#include <tli.h>* statement is needed at the beginning of the program.

4.1.2 Implementing a conventional library for TLI functions Whatever the application is, either a client or a server, it first calls the function *t_open* to create a new communication descriptor that might be further used for network communications. Arguments to *t_open* specify the transport provider (i.e. the protocol software that should be used), the desired mode (e.g. whether the descriptor will be used to read or write data) and some information about communication in the *t_info* structure. For example the field *servtype* specifies whether the communication descriptor to be created uses TCP or UDP. An implementation of the function *t_open* is shown in Figure 3.

```
short t_errno,errno=0;        //global variables
com_descriptor t_open(char FAR *path,int oflags,
                              t_info FAR *info)
    {short sh;
    if((info->servtype < T_CLTS)  // UDP client or server
    ||(info->servtype < T_COTS))  // TCP client or server
          {t_errno=TSYSERR;
          return(-1);}
          asm cli;
          asm pusha;
          asm mov bx,oflags;
          asm mov ax,seg path;
          asm mov ds,ax;
          asm mov si,offset path;
          asm mov ax,seg info;
          asm mov es,ax;
          asm mov di,offset info;
          asm mov ah,29h;   // allocate global descriptor
          asm int 61h;
          asm mov sh,ax;
          asm jnc no_error;
          t_errno=TSYSERR;
          errno=sh;         // sh = error code
          asm popa;
          asm sti;
          return(-1);
no_error: asm popa;
          asm sti;
          return(sh);       // sh = network descriptor
          }
```

Figure 3: Example of t_open implementation (see Figure 1)

Transactions on Information and Communications Technologies vol 10, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

All the other TLI functions, written in the same way like *t_open*, must be added in the file *tli.c*. To create a conventional library, after compilation a librarian must be run instead of the classical linker. The result will be a library file, *tli.lib*, that will contain the implementations for every TLI function. Whenever a C application needs to call a TLI function, this function is statically linked after compilation by the linker and the body of the needed function will be part of the executable. Therefore, as Figure 4 suggests, if more applications run simultaneously, more copies of the same functions will be found in the memory at the same time.

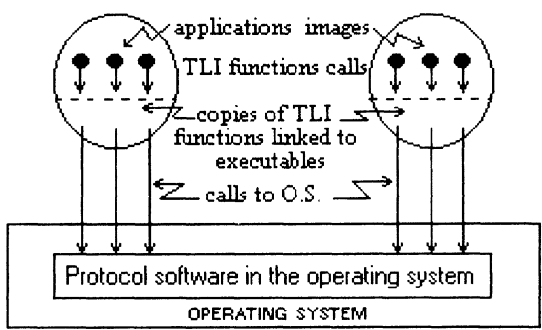

Figure 4: Static linked library implementation of TLI functions

4.1.3 Implementing a shared library for TLI functions Even if a communication descriptor is already created it does not have either local or remote endpoint addresses specified. Thus both clients and servers call *t_bind* to specify the local endpoint address for the given communication descriptor.The arguments to *t_bind* specify the communication descriptor and two pointers to *t_bind* structures: one is called *request* and the other one *result*. The pointer to the *request* structure might be NULL, showing that the caller does not care about the endpoint address which will be chosen by the service provider. Otherwise the *request* structure will contain the endpoint address and the maximum number of TCP connections the system will queue up. All the other TLI functions have to be implemented in a similar way. Then the resulting *tli.c* file will be compiled by a C for Windows compiler, building a DLL. For this, instead of a WinMain entry point for the executable, rather a LibMain entry point will be used. Whenever a new application is to be created it might use all the functions, structures, global variables and predefined symbolic constants assuming that the *tli.h* header file is included and all the functions used are declared as 'IMPORTS' in its *.def* file. Then, when the first application that uses TLI functions is to be run, the DLL file is loaded and dynamically linked to it. Each other application will use the same, already loaded, DLL. Therefore, even if there are more applications that are running simultaneously, they will share the same copy of the TLI functions code and only one set of functions might be found in the memory (Figure 5). However,

Transactions on Information and Communications Technologies vol 10, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

these functions are related to different data segments when called and thus they will provide the appropriate results to the caller [4].

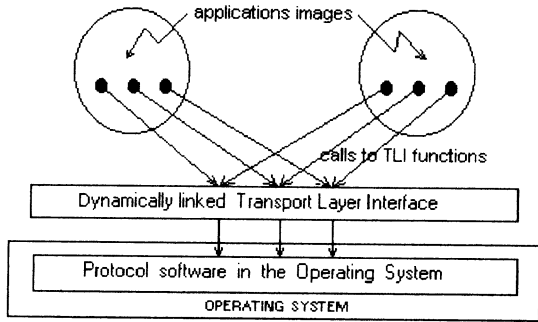

Figure 5: Dynamic linked library implementation of TLI functions

4.2 UNIX version of a client-server architecture

The teaching experiment is assumed to include a UNIX version of a client-server architecture, but without TLI functions. Based on students' reaction to the DOS/Windows version previously discussed, it seems that, for better understanding of the terms (such as socket, port, client, server), other point of view could be useful. Some teaching problems encountered by use of a new symbolic concurrent programming language Erlang for telecommunications applications, have been presented at SEHE 94 by Dobrota [5]. Underlining the advantages compared to conventional imperative languages, a rapid implementation of a client-server architecture is presented. It is assumed that there are two processes and an inter-processes communication using a socket.

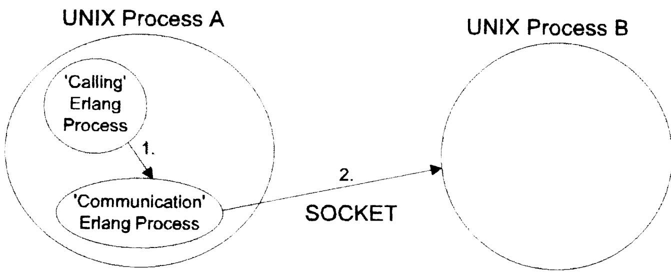

Figure 6: Inter-processes communication using a socket

The following examples present the mechanism of starting a socket server and a socket client (Figure 7). Then it is not very difficult to test the output and input for both processes. Obviously the implementation of TLI functions in Erlang could be for further study, as well as other combinations of programming languages and operating systems: C++ for UNIX, Erlang for DOS/Windows etc.

Transactions on Information and Communications Technologies vol 10, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517


```
server_socket (Name) ->
      socket:start(),
      ListenSocket=socket:listen('STREAM','AF_INET',0,
                                    twobytes),
      {FD, Port} = ListenSocket,
      io:format("~s:attempting to use port:~w~n",[Name,Port]),
      Accepted = socket:accept(ListenSocket),
      io:format("~s: connected to port: ~w. Id = ~w~n",
                                    [Name,Port,Accepted]),
      Accepted.
```

```
client_socket (Name, Machine, Port) ->
      socket:start(),
      io:format("~s: connecting to port: '~w' on machine
                  '~w'~n", [Name,Machine,Port]),
      Socket=socket:client('STREAM','AF_INET',{Machine,Port},
                                                   twobytes),
      io:format("~s:connected to socket:'~w'~n",[Name,Socket]),
      Socket.
```

Figure 7: Starting a socket server and socket client in Erlang under UNIX

5 Conclusions

A teaching experiment has been launched in order to find the suitable methods to teach client-server architecture and their particular programming techniques. The first approach was from standpoint of a future software engineer working in telecommunications and setting up a DOS/Windows version, internetworking with TCP/IP. TLI functions were assumed to be used, implementing conventional or shared library for them, written in C++. The second approach was from standpoint of a future communications engineer designing a software tool. The UNIX version of a client-server architecture, but without TLI functions, has been done in Erlang, a new symbolic concurrent programming language.

References

1. Comer, D. & Stevens, L.D. *Internetworking with TCP/IP - AT&T TLI version*, Vol.III, Englewood Cliffs, Prentice Hall, 1994.
2. Stallings, W. *SNMP, SNMPv2, and CMIP. The Practical Guide to Network-Management Standards*. Addison Wesley Publishing, 1993.
3. Berson, A. *Client/Server Architecture*. McGraw-Hill, Inc.1992.
4. Pietrek, M. *Windows Internals. The Implementation of the Windows Operating Environment.* Addison-Wesley Publishing Company, 1993.
5. Dobrota, V. Use of the Symbolic Concurrent Programming Erlang to Test the Integrated Digital Networks Access, SEHE 94, in *Software Engineering in Higher Education SEHE 95 Proceedings.*