ADDCE

Rough At The Edges

A Distributed System

designed & written
by

Tobias Mayer
South Bank University
1997

Contents

		page
Section 1	**"Rough At The Edges" Defined**	2
Section 2	**The Protocol**	3
Section 3	**The Generic Module**	5
Section 4	**Description of the socket system calls**	7
Section 5	**The RATE Applications**	10
Appendix A	**The "Real-Life" Cycle**	12
Appendix B	**Sources & References**	13

"Rough At The Edges", hereafter referred to as RATE, is an Application Layer Communications Protocol that is defined on TCP/IP. It is the protocol used by the RATE system.

The RATE system consists of a generic client & server pair and a suite of user application programs that 'slot in' to generic the module.

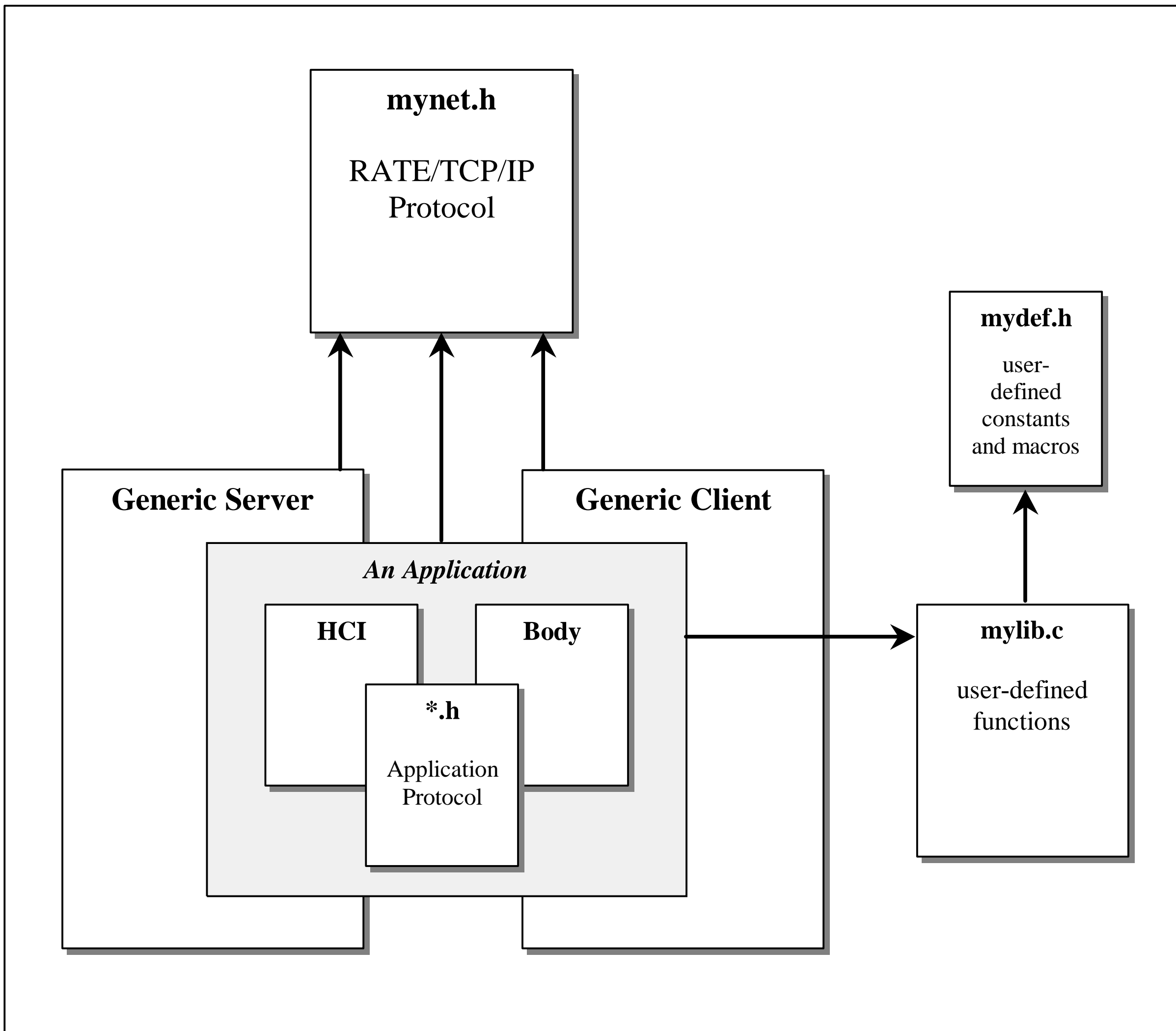

Figure 1.1
Logical model showing:
1. *Separation between the Generic System and the Application Modules - different modules can be inserted, each requiring its own *.h file and possibly using **mylib.c**.*
2. *Distribution of source code into logically separate files.*

RATE is defined on TCP/IP. This was chosen in preference to UDP for reasons of integrity and reliability.[1]

The Internet protocol (IP) itself is located at the Network layer of the OSI model. It provides a connectionless and unreliable delivery system. Both TCP and UDP sit on top of IP at the Transport layer of the OSI model. IP expects the upper layers to take responsibility for reliability of delivery and error checking.

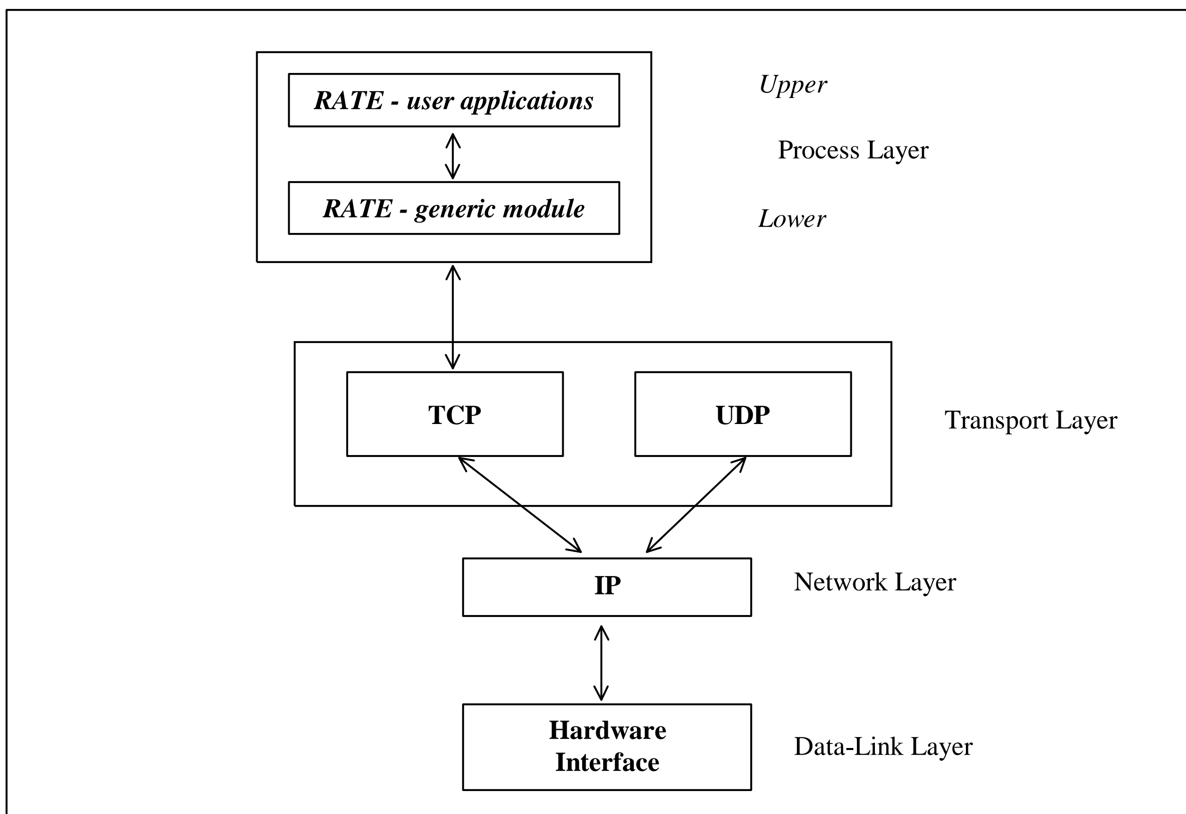

Figure 2.1. The relationship between IP, TCP (& UDP) and RATE on the simplified, 4-layer, OSI model (the Upper/Lower distinction at the Process Layer is mine; it is used to distinguish between the generic module and the application programs that use/sit on the generic module).

UDP provides only two facilities not available at the IP layer: the use of port numbers and an (optional) check sum to verify the contents of a UDP datagram. UDP is also connectionless and unreliable. TCP, on the other hand, provides a connection-orientated and reliable delivery service which includes such features as positive acknowledgment, a 'timeout' service and duplication detection. The initial implementation of a TCP client/server pair is slightly more complicated than the implementation of UDP, (e.g. a TCP server requires 'listen' and 'accept' calls whereas UDP does not) but this is superficial, as no additional verification/ validation code needs to be written for the TCP implementation.

[1] Other protocols such as those defined by Xerox NS and Unix were not considered as my knowledge of them does not extend beyond knowing of their existance.

TCP does use more CPU time due to the checking it carries out, so if a large number of TCP client/server applications are running at one time the system could slow down. But again this is of minor consideration.

South Bank University is on a LAN (Local Area Network) so much of the security and validation lacking in UDP is provided by the underlying Data Link layer, using Ethernet technology. It would therefor have been acceptable - or even preferable - to use UDP with no additional user-defined validation/checking if it was known that the RATE applications were only to be used from within the LAN. In reality, of course, this is the case, but ideally I wanted to make the RATE suite of applications more widely usable. Once the applications are running across a WAN (Wide Area Network) the support provided by Ethernet no longer applies. Many different network architectures are likely to be used between client and server on a WAN.

Section 3 The Generic Module

Server

The RATE generic server is designed as a concurrent server and is configured (using a set of *for loop* structures) to accept one or many clients for a particular session. The number of clients is determined by each user application program and is defined in the application's header file as:

```
#define NO_OF_CLIENTS    n
```

The server uses the principle of connecting one client to one port number to overcome the problem of linking multiple clients. The other way of doing this would be to use a single port and implement the 'select' call. The multi-port number method is simpler to implement and allows the server to easily identify a client by the port number it is connected to. It would be difficult to send data to the wrong client in a multi-client session. If the RATE system was intending to link 100's to 1000's of different clients this would not be a suitable method, but as it is intended to handle only two or three, or maybe a dozen clients at a time there is no danger of running out of port numbers.

```
for each client do
      create a socket
      fill in a sockaddr_in structure:
            family = AF_INET (TCP/IP)
            address = ANY (ie accept clients on any address)
            port = next port # in sequence
      bind (socket to port)
end;

for each socket do
      listen
end;

do
      for each client do
      //server stays in this loop until all clients accepted//
            accept client
            write 'Validate' message ✍
            ✍ read name of user application
      end;

      if all clients running same application
            --< fork child process & call appropriate user function.
      else
            for each client
                  disconnect socket
            end;
      end if;

loop indefinitely.
```

Figure 3.1. Basic structure of the RATE Generic Server

Client

The RATE generic client connects to the generic server. It contains the `main()` function which calls the user application function/s as appropriate. It is designed to run multiple copies of itself depending on the number of users connecting in the user application. The generic client is responsible for ensuring that the user application is:

- connected to the correct server
- running the same application as the other user/s connected in the same session.

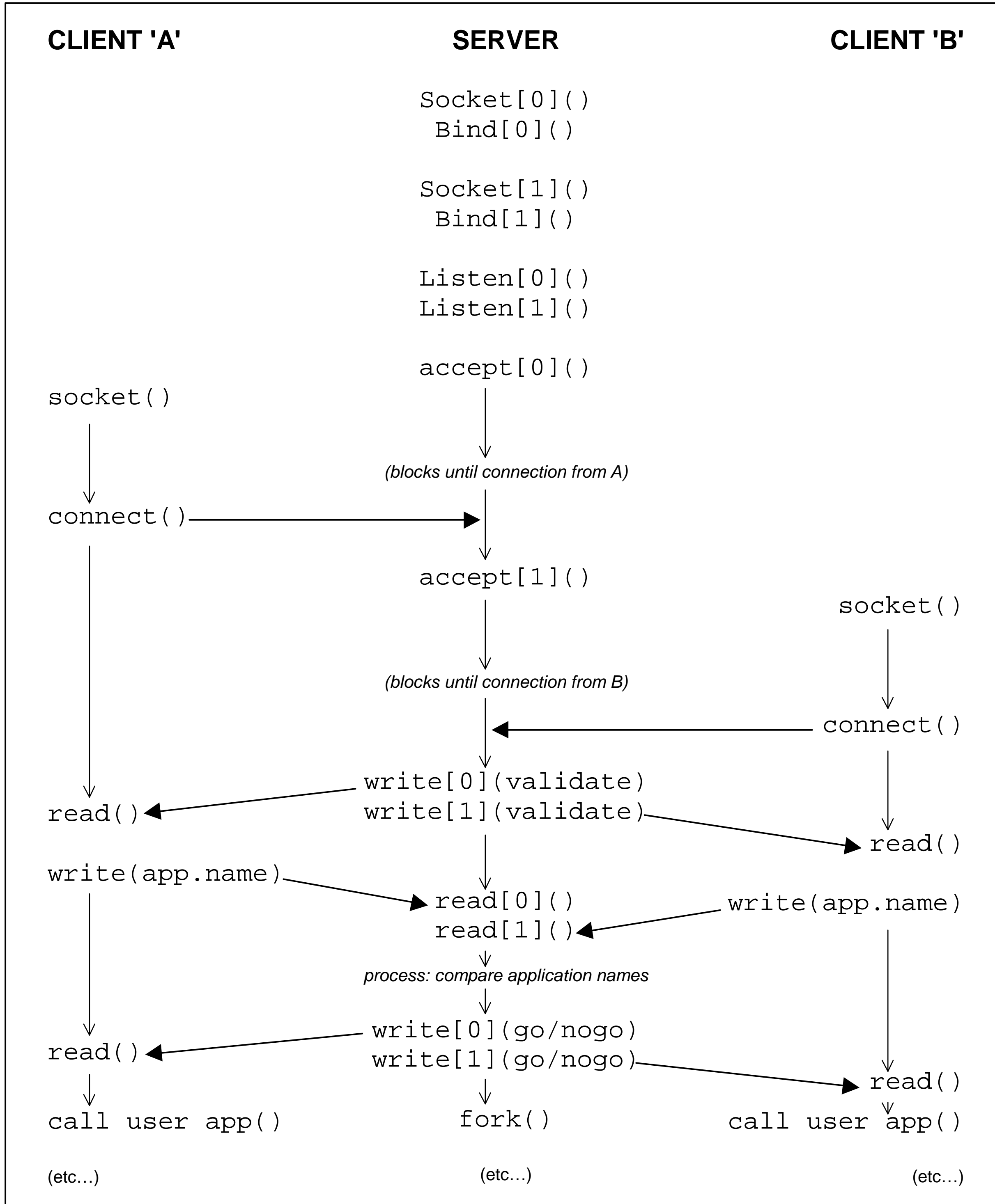

Figure 3.2 Relationship between Generic Server and two Generic Clients

Section 4 Description of the Socket System Calls

The reader should refer to the source code files to see the actual parameters passed to the following system calls. In this report I will discuss them in a more general way.

`socket`

The `socket` call is made by both server and client. It requires three integer parameters***:*** ***family***, ***type*** and ***protocol***. The values for these are defined as constants in the header file <sys/socket.h>.

The ***family*** refers to a set of network layer protocols, namely
- IP (Internet)
- Xerox NS
- IMP Link Layer
- Unix.

The ***type*** is determined to some extent by the family as not all combinations of ***family*** and ***type*** are acceptable. The ***type*** is one of:
- Stream
- Datagram
- Raw
- Sequenced Packet

The Internet family can use the first three of these types. The ***family*** and ***type*** combined give the ***protocol***. Because the protocol can be worked out from the first two parameters the actual value for the third parameter is often given as 0, but it is the actual protocol that is specified in the data structure that is created as a result of this call.

This data structure is stored in the Operating System and the `socket` call returns a small integer value known as a socket descriptor which refers to this data structure. The other elements of this data structure:
- Local address
- Local process
- Foreign address
- Foreign process

remain unfilled at this stage. In order that the socket descriptor be usable these will need to be specified by subsequent calls.

The system calls following `socket` depend on whether the protocol used *is connection-orientated* (e.g. TCP) or *connectionless* (e.g. UDP) , they also differ between client and server -in both instances. For the purpose of this report I will concentrate on the protocol being used by the RATE system: namely TCP/IP.

Before the next system call both client and server need to specify a 'server structure' (`struct sockaddr_in`). This requires definitions for the protocol family (as above), the address of the system where the server is running and the port or process number. The latter two are network byte-ordered, meaning that they need to be translated to the byte order of the system that the server is running on when extracted.

If the address (as dotted decimal) is not known it can be located by the use of the `gethostbyname()` system call. This call takes a system name as a parameter and fills out

a `hostent` structure with the address of the system, the size of the address and any aliases it is known by.

Once the address and port number are defined the server can make a call to `bind`.

`bind`
This call assigns a name to an unnamed socket. It takes the socket descriptor as its first parameter and fills out the *local address* & *local process* elements of the data structure which it extracts from the server structure given as the 2nd parameter. (The 3rd parameter is the size of this structure.) Following the bind call the data structure is considered to be a *half-association*. The other *half-association* will be obtained from the client process. (Both *half-associations* contain the protocol) .

`listen`
The server calls `listen` to indicate that it is ready to receive connections at the given socket. The second parameter for listen is a number between 1 and 5 that determines the queue. In the time it takes for a server to handle the accept request and fork a child process there may be a backlog of clients waiting to connect. Without this queue they would be rejected.

`accept`
Once the listen call has been executed the server goes into an indefinite loop where it accepts connections and forks child processes to handle them. The `accept` call creates a second data structure and socket descriptor identical to the first. The 4th and 5th elements of the data structure can then be filled in with the address and process id of the client process. It is this second socket descriptor that is used by the child process.

`connect`
The client calls `connect` following its `socket` call. `connect` establishes a connection with the server. The parameters are the same as those for `bind` but because the client has already been given the server address & port number (either on the command line or as fixed values in the header file) this call is able to fill out the four remaining elements of the socket data structure, namely *local addr, local process, foreign addr, foreign process*. The client process could use a `bind` call to bind a local address before calling `connect` and there may be times when it is useful to do so, but on the whole it is unnecessary .

Other calls made by the generic client & server

Byte Ordering Routines
Both the address & port number elements of the `sockaddr_in` structure need to be stored as network byte-ordered values. To carry out this conversion the routines `htons` and `htonl`, (host-to-network) handling short and long integers respectively, are used. Two sister functions, `ntohs` & `ntohl` carry out the reverse operation.

Byte Operators

The system calls `bcopy` and `bzero` are available to deal with non-'C'-type strings, i.e. strings that do not use the null character as a terminator. Both functions require a ***length*** parameter to determine the number of bytes to be handled. `bcopy` moves the specified number of bytes from the source address to the destination and `bzero` writes a null character to every specified byte. The latter is also useful when using C-strings as it ensures that a null character will be the last byte of any string copied or transferred and avoids the problem of not sending or copying the last byte of a C-string.

Interaction with the System

The generic client uses the `getenv` function to obtain the user's user-id from the system. The call can obtain any of the system's environment variables. The client also makes use of `system` to temporarily exit to the unix operating system and carry out a `clear` command.[2]

---

[2] The `system(clear)` call is done from a child process. The reason for this is that I intended to use the Unix `exec` call (which requires a child to be forked) but consistently received an 'incorrect address' message when attempting to compile. The use of `system` is a compromise, but it does the job equally well.

Section 5 The RATE Applications

Three applications have been designed and implemented for the RATE system:

? **Prisoners' Dilemma** - a strategy game for two people
? **A Very Basic Conference Program** - one-to-one chit-chat
? **Text File Transfer** - transfer of ASCII files of up to 1 kilobyte

Each of these requires two clients to be connected to the server. The source code for each application is divided between an HCI file and a body/functionality file. The Application protocols are defined in the application's own header file. The body file contains the functions for both client and server. Figure 1.5 shows the #include relationships between the source files.

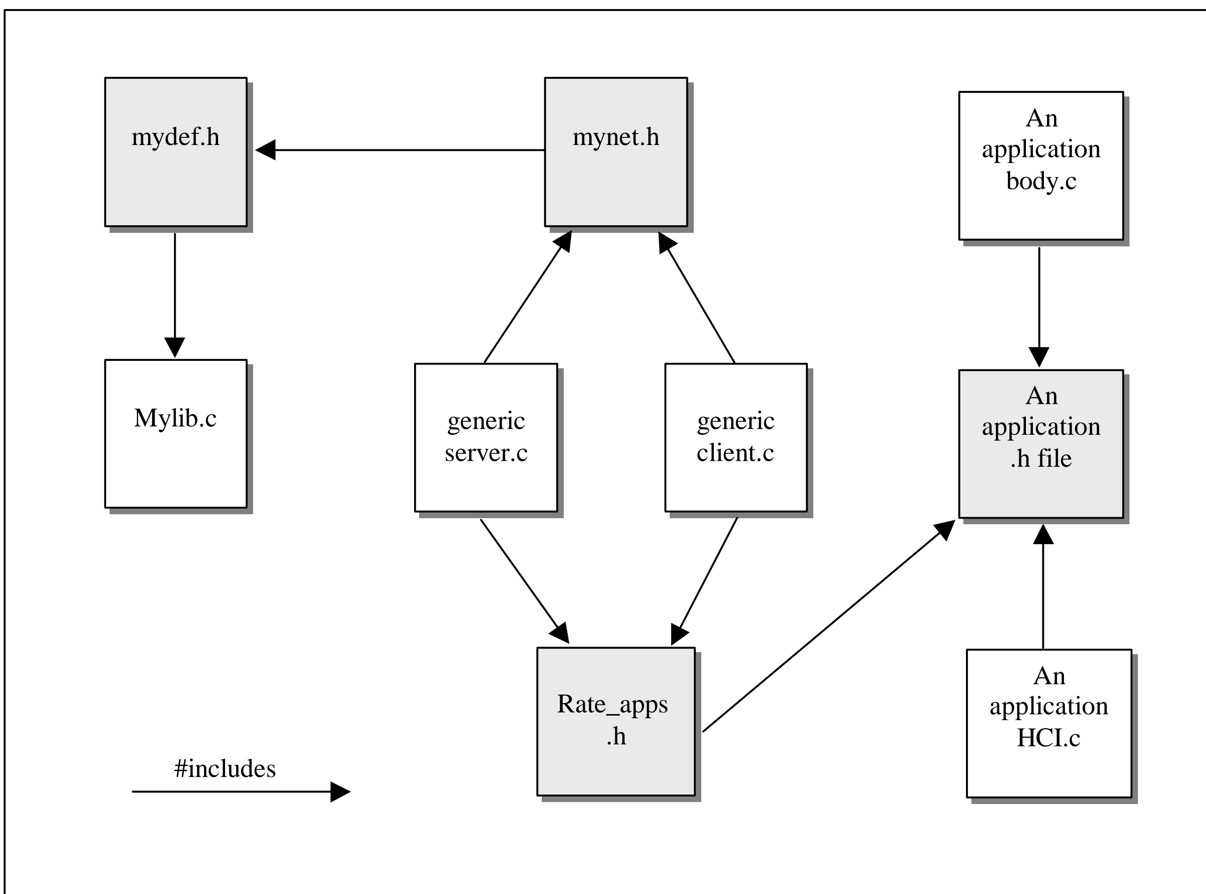

*Figure 5.1 #include relationship between *.h & *.c files*

JSP Designs are included for Prisoners' Dilemma & Conference. Text File Transfer was a late addition to the suite, and was quickly hacked together; thus no designs are available.

The previous statement indicates that a Software Engineering approach was nowhere in sight during this development. This is correct. My lack of knowledge of the C language and of Networking meant that the system was developed in a hacking, experimental way with goals only becoming clear after a good deal of experimentation.

The JSP's were done at the end, and taken from the source code files. The reason for producing them at all is for maintenance and further development purposes. The source code files for all three are also well commented throughout.

I implemented a number of standard library functions, which are grouped together in the file 'mylib.c'. Among these is an Error Check function that I use to wrap around all of the 'read' and 'write' calls. This saves re-writing the same four or five lines of code for every call. Another function I particularly like is 'manyCat' which takes a variable number of parameters of type *char and concatenates them into a single string. This saves calling 'strcat' something like 16 times in succession. The use of these library functions has, I believe, greatly enhanced the readability - and therefor the maintainability - of the code.

The role of the server in the three application programs differs. In **Prisoners' Dilemma** it carries out all the processing on behalf of the two clients and returns the results to them. In **Conference** & **TFT** the server simply acts as a go-between, passing data between one client and another.

Little more needs to be said about the application programs. They speak for themselves. They work. They prove the point that the generic module really is generic.

Design for a realistic Software Development Life Cycle, suggested by D.L.Parnas and P.C. Clements in the article "A Rational Design Process: How and Why to fake it", IEEE Trans. On Software Engineering., vol 12, no 2, February 1986, pp251-257.

A plan not dissimilar to this, from the specification onwards, was followed for this coursework.

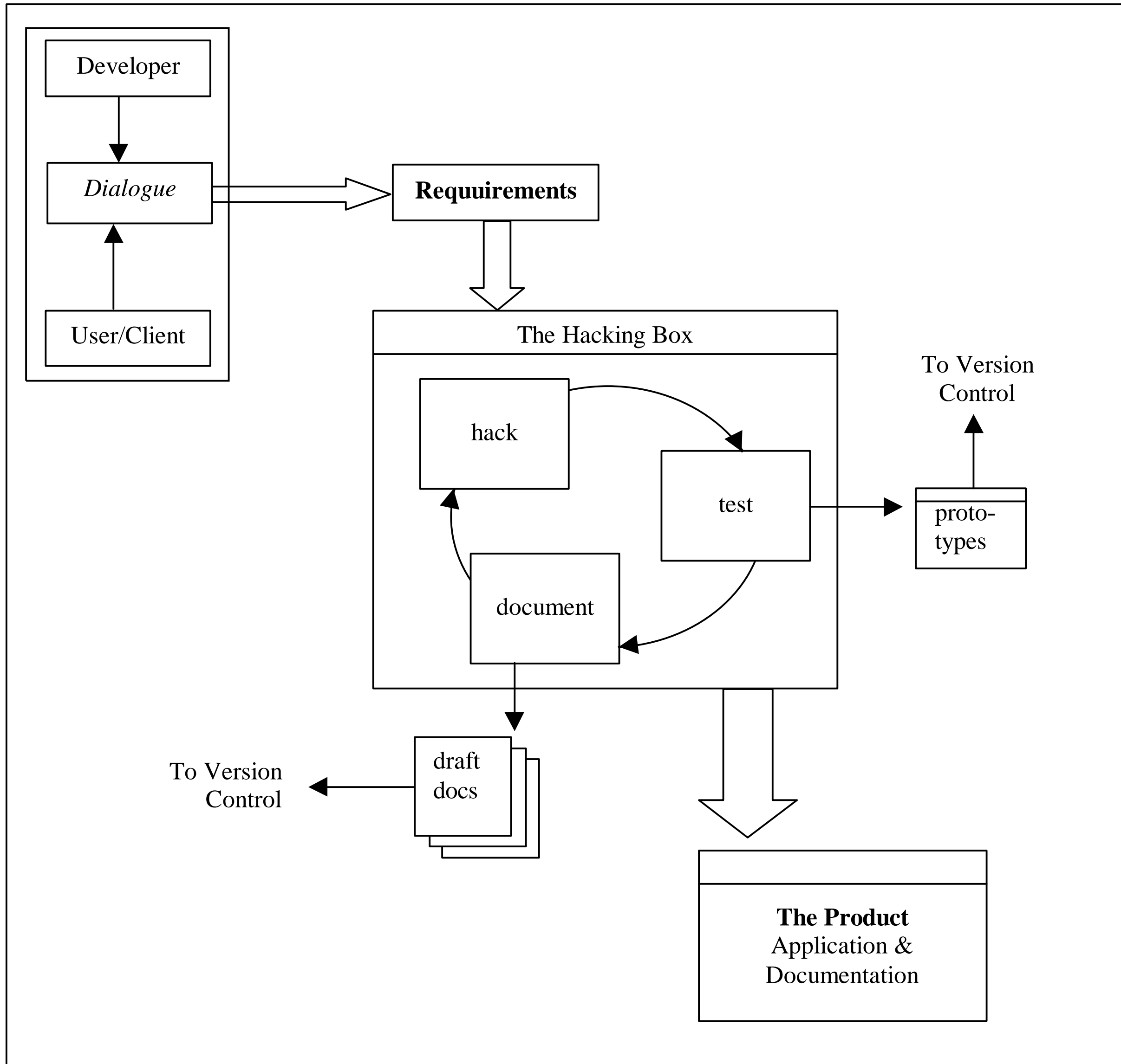

Figure A. The "Real-Life!" Cycle

Appendix B Sources & References

Unix Network Programming	W. Richard Stevens 1990 Prentice Hall Software series
C For Programmers, 2nd Edition	Lendert Ammeraal 1991 John Wiley & Sons
ADDCE Course Notes	Mike van Kleef 1996 Southbank University, SCISM
Unix in a Nutshell, 2nd Edition	1992/4, O'Reilly & Associates, Inc