Report Number: WUCS-95-30

1995-01-01

Design of a Tool for Rapid Prototyping of Communication Protocols

Aniruddha Gokhale, Ron Cytron, and George Varghese

We present a new tool for automatically generating prototypes of communication protocols on a wide variety of platforms. Our goal is to reduce design time, enhance portability, and accommodate optimizations automatically. Users of the tool are required to provide an abstract implementation of the protocol in C++ without worrying about the underlying operating system specific system calls. Instead, the user employs high-level interface functions provided by the tool to interact with the underlying operating system. Users also need not worry about complex packet formats that involve fields of various bit and byte lengths. Instead, they use simple C/C++ struct... **Read complete abstract on page 2.**

Design of a Tool for Rapid Prototyping of Communication Protocols

Aniruddha Gokhale, Ron Cytron, and George Varghese

Complete Abstract:

We present a new tool for automatically generating prototypes of communication protocols on a wide variety of platforms. Our goal is to reduce design time, enhance portability, and accommodate optimizations automatically. Users of the tool are required to provide an abstract implementation of the protocol in C++ without worrying about the underlying operating system specific system calls. Instead, the user employs high-level interface functions provided by the tool to interact with the underlying operating system. Users also need not worry about complex packet formats that involve fields of various bit and byte lengths. Instead, they use simple C/C++ struct declarations to describe the packets and provide mapping rules in the form of Extended Regular Expressions to the tool. The tool uses these rules to convert between the network format and the user defined format. Experience with TFTP and SMTP prototypes indicate that the performance achieved is almost comparable to that of the standard BSD implementations; at the same time the code size requirements of the abstract implementation is roughly 3 times less than the BSD implementation.

Design of a Tool for Rapid Prototyping of Communication Protocols

Aniruddha Gokhale, Ron Cytron and
George Varghese

WUCS-95-30

October 1995

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis, MO 63130-4899

Design of a Tool for Rapid Prototyping of Communication Protocols

Aniruddha Gokhale*(`gokhale@cs.wustl.edu`) Ron Cytron (`cytron@cs.wustl.edu`)
George Varghese (`varghese@cs.wustl.edu`)

Department of Computer Science
Washington University Box 1045
St. Louis, MO 63130
Phone: (314)935-6160 Fax: (314)935-7302

October 16, 1995

Abstract

We present a new tool for automatically generating prototypes of communication protocols on a wide variety of platforms. Our goal is to reduce design time, enhance portability, and accommodate optimizations automatically. Users of the tool are required to provide an abstract implementation of the protocol in C++ without worrying about the underlying operating system specific system calls. Instead, the user employs high-level interface functions provided by the tool to interact with the underlying operating system. Users also need not worry about complex packet formats that involve fields of various bit and byte lengths. Instead, they use simple C/C++ *struct* declarations to describe the packets and provide mapping rules in the form of *Extended Regular Expressions* to the tool. The tool uses these rules to convert between the network format and the user defined format. Experience with TFTP and SMTP prototypes indicate that the performance achieved is almost comparable to that of the standard BSD implementations; at the same time the code size requirements of the abstract implementation is roughly 3 times less than the BSD implementation.

Keywords: Communication Protocols, Prototyping, Tools, Compiler Technology.

1 Introduction and Motivation

Networks and applications are getting more diverse to span a wide range of technologies, platforms and user needs. The problem of designing network software, already a difficult task, is exacerbated by the increasing diversity. For example, there are several standards for high speed Data Links[1] and most router vendors support at least six routing protocols. [2] At the same time network software continues to be hard to implement and maintain. Dealing with asynchronous events, obscure race conditions, and poor support for distributed debugging make even user-level protocol implementations hard to develop.

On the other hand, the field of compiler design has matured to the point that a wide-range of *code generation* techniques exist to generate optimized machine code from programs specified in higher level languages. In fact, code produced by compilers is often better than or competitive

*contact author

[1]e.g., FDDI, HIPPI, Fiber Channel, ATM

[2]IP, OSI,DECNET, SNA, Appletalk, and Novell Netware

with hand-crafted code. For example, code on RISC machines requires careful attention to inter-related, low-level details (such as register allocation and filling delay slots) that are best left to the compiler. The success of RISC machines is largely due to the availability of good compilers that free the programmer from the details of the machine architecture. Thus it is natural to ask: **can compilation techniques be applied to networking software?**

Performance is also critical for crucial parts of the code (e.g., transport protocols like TCP). However, the amount of performance critical code is typically quite small. On the other hand, the amount of management code for most protocol suites is several times larger than the code that actually implements the protocol. Similarly, there will probably be many more application protocols emerging that take advantage of the networking infrastructure. For many such protocols ease of implementation and maintenance can be traded off against some small loss of performance due to the use of compiler tools. For those protocols for which performance is crucial, however, we note that automatic optimization techniques can be incorporated into network compiler tools.

Our long-term solution to the problem of designing network software consists of two main parts: a tool for rapid prototyping of networking software that compiles high level implementations of protocols to a variety of different "back ends": this is our proposed solution to the problem of heterogeneity, and is described in the paper. The second part is a systematic development of automatic optimization techniques for network protocols: this is our proposed solution to the need for high performance. We leave this for future work, though we describe a few ideas in Section 4.

1.1 Our Tool

Our tool can be used for *rapid prototyping* and *easy maintenance* of communication protocols. We assume that the protocol specification is complete and available to the tool user. The tool user then writes an abstract implementation (that we call the *Protocol Essence*) without regard to the specific platform the implementation will run on. The abstract implementation is written in a "friendlier" (as detailed below) environment than the target platform, which allows the implementor to write a smaller amount of code in (hopefully) shorter time.

Note that our approach is entirely different from approaches that attempt to compile an implementation directly from a protocol specification. A fully abstract protocol specification specifies *what* a protocol should do and not *how* the protocol satisfies its specification. While this approach is an area of active research, we do not know of any successful application to real protocols. Instead, our abstract implementation or protocol essence actually describes *how* the protocol implementor satisfies the specification, except that target-specific details are abstracted away.

Note that due to the difficulty of providing truly abstract protocol specifications, the documentation for most real protocols (e.g., TCP, OSI TP-4) provide a considerable part of what we would call an abstract implementation. Some documentation (e.g., the 802.1 bridge specification, OSI TP-4) even provides what is essentially an abstract implementation in pseudo-code or C! Thus there is considerable motivation for a tool that can leverage off the existing documentation to directly provide a real implementation.

The salient features of our tool can be summarized as follows:

- Only the abstract implementation of the protocol needs to be provided:

  An abstract implementation of a protocol may be a faithful or an approximate implementation of the protocol specification. It need not worry about any specific operating system features or errors resulting from failed system calls. Instead, it uses high-level functions provided by the tool to interface with the underlying operating system. Recall that we call such an abstract implementation the *Protocol Essence.*

- Protocol essence can be authored without undue attention paid to:

  Wire data formats: user code primarily references simplified packet formats (C/C++ structures); our tool automatically integrates the separately specified wire formats.

  OS demultiplexing: (*e.g., Poll* or *Select*) is added automatically.

  User interface code: we employ a YACC-based interface generator to automatically generate user interface code based from the (YACC) specification provided by the user.

  Error Handling: system calls provided by the underlying operating system may fail and return obscure error codes. In the current form, the tool's library provides a mechanism by which failures of these kind are detected and the session is terminated. In future, we would like to look at efficient exception handling mechanisms.

- We can target a surprising range of environments, including a simulator, and application-level network interfaces.

1.2 Organization of Typical Networking Code

Network software is generally developed in languages like C. These languages provide type casts, pointer arithmetic, macro definitions, global variable declarations, etc., which allow for high efficiency. Network software written in C invariably exploits these features to the maximum extent possible.

In addition to the prolific use of these language features, the software also blends the protocol specific code, the error handling code and the underlying operating system dependent code. This tight coupling makes it extremely difficult to comprehend, debug and maintain network software. Figure 1 depicts a typical view of such a network protocol software. The dark rectangles denote protocol-specific code, the lightly shaded ones denote error handling code and the blank rectangles denote operating system dependent code that may deal with interrupt processing, buffer manipulation, input-output, scheduling, etc.

On the other hand, separating the protocol essence, the error handling code and the operating system code has several advantages:

- It improves the comprehensibility of the code.
- Debugging efforts expended during the implementation phase can be significantly reduced.
- It is possible to observe and test the protocol in isolation. This could be used in the process of protocol conformance testing and further refinement of the protocol specification.

1.3 Use of the tool by Protocol Designers

The primary set of users we target for the tool are protocol implementors. However, the rapid prototyping and simulation aspects of our tool can also be used by protocol designers developing a new protocol. Figure 2 depicts how such a tool can be put to use by a protocol designer.

- The protocol designer can choose to refine the original protocol specification incrementally. The protocol implementor uses the protocol specification to write the protocol essence, which in turn can be provided as input to the tool. The tool can generate a simulation which can

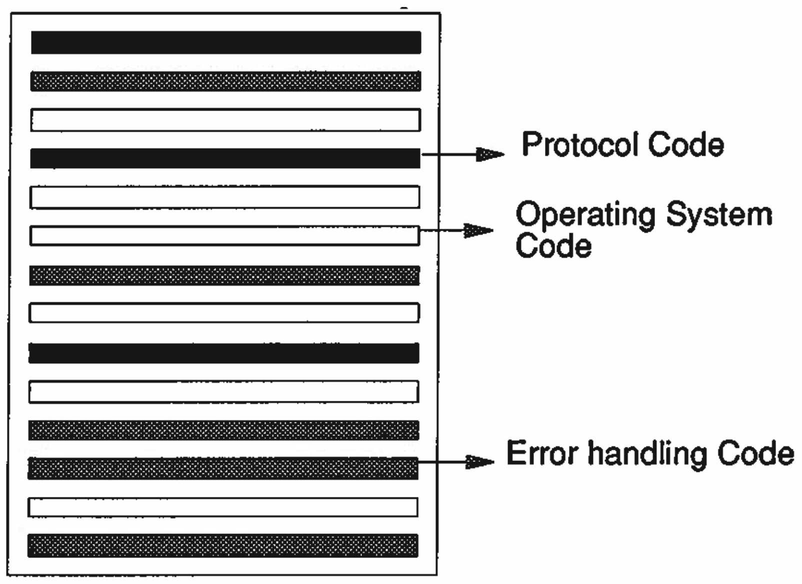

Figure 1. Typical Protocol Implementation

help the protocol designer observe the behavior of the protocol without any interference from the operating system. These results may be used to further refine the specification.

- Once the protocol design is stable and is tested for conformance, a prototype could be rapidly produced and effects of external events on the protocol could be observed.
- In the event that some changes need to be made to the original protocol design, the only overhead incurred is to transform the modified protocol specification into a protocol essence. The tool can rapidly produce the simulation and the prototype saving considerable effort that would have otherwise been spent in hand-crafting a simulation and a prototype implementation.

We note again that due to the difficulty of writing truly abstract specification, many protocol designers essentially provide an abstract implementation of their protocol. Thus the step of converting the specification to the protocol essence is often easier than one might think.

1.4 Complex Message Formats

In addition to the protocol's intricate event handling mechanism, complicated message formats make the task even more difficult and susceptible to errors. These message (packet) formats may consist of a large number of bit or byte fields of varying length. Software implementations that directly make use of such physical formats for setting and retrieving values of various fields invariably lack portability. What makes things particularly complicated is the problem of byte and word ordering within packet fields.

Some machines are *big endian* machines (that number their bytes such that byte 0 is the high-order byte); other machines are *little endian* that number their bytes the other way. Finally, protocols mandate the byte and bit order of fields as they must appear on the wire. As a result,

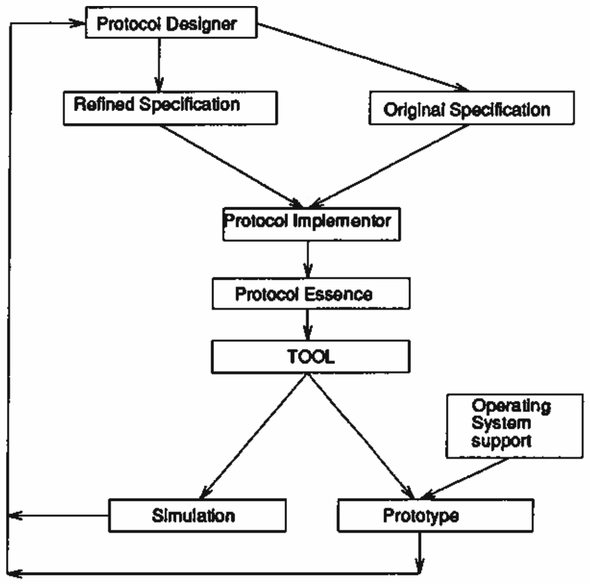

Figure 2. Use of Tool by Protocol Designers

most implementors have to manually do a considerable amount of byte and word swapping[3]. This is time consuming, makes portability difficult, and is also error-prone. From our experience, a number of implementation bugs are caused by confusion about endianness.

Owing to these factors, it is important to find new ways for implementors to manipulate packet fields. Our approach is to allow the implementor to view the packet formats at a logical level. At this level the fields are declared using simple structure types provided by the implementation language. This allows the implementation to be portable and also improves readability and maintainability of the code. Tools serving as back-end compilers can be built to convert between the actual physical format and the logical view of the packet formats.

Related Work

A number of tools for protocol prototyping or simulation appear in the literature.

We first distinguish our approach (compiling abstract implementations) from what we call the *virtual machine* approach. The virtual machine approach, as exemplified by the x-Kernel [4], raises the level of abstraction of network-level programming without taking advantage of compile-time analysis. This is done by providing a suite of useful library functions for many common protocol tasks. Our approach is more general because it allows compile-time analysis based on a complete scan of the program text. We note also that environments such as the x-Kernel can serve as targets of our work.

Another example of the virtual machine approach is the ACE-ASX [12, 11, 10] framework, which provides an environment for constructing application-level protocols. It provides many features to incorporate operating system synchronization features into the protocol implementation separately. The ACE-ASX framework provides for encapsulating operating system interprocess communication

[3]sometimes, as in UNIX, this is done using library calls but the protocol implementor has to pepper his code with such calls; in our approach the code that specifies the byte conversion is segregated to one area.

mechanisms within object oriented wrappers. These wrappers provide a generic interface to the underlying operating system. [4]

Another approach is to build tools that compile protocol specifications into implementations. An example is the PET-DINGO tool suite [14, 13] that build prototype implementations of protocols from specifications provided in the Formal Description Technique - Estelle. Our approach differs from the Pet-Dingo approach in that our tool produces prototypes from abstract implementations as opposed to specifications. Also our abstract implementations are authored in C++. From our experience, most implementors are more comfortable in C/C++ than in Estelle.

Some work related to encoding packet formats is also reported in the literature. The External Data representation (XDR) standard [6] is used for the description and encoding of data and is used in the RPC protocol. The Universal Stub Generator (USC) [8] describes a stub compiler that generates stubs to perform many data conversion operations. However, they do not provide a complete tool that meets all our goals.

One aspect of our tool that is extremely useful is the ability to also build simulations from the abstract implementation. While prototypes can also be used to simulate a protocol, a single process simulation of a N-party protocol with debugger support, is a far more convenient and repeatable way to observe protocol dynamics. The user has to provide the abstract implementation and the network topology, and the tool outputs a discrete event simulation of the protocol on the given topology. The literature on network protocol simulation is too prolific to report here. For example, the Nest [2] tool provides a tool for simulating distributed systems. However, most simulators that we know of are not capable of producing prototypes.

1.5 Organization

This paper discusses the design considerations and protocol prototyping experiences with such a tool. Simplified versions of the TFTP protocol [15] and SMTP protocol [9] have been chosen as examples for this paper. The rest of the paper is organized as follows.

Section 2 describes the system overview and design methodology. In Section 3, our experiences implementing the TFTP and SMTP protocol have been reported. Section 4 discusses advantages and disadvantages of using such a tool and also discusses future development on this tool.

2 System Overview and Methodology

We now present an overview of the system and describe the design methodology used in building the tool. We begin by providing an overview of the system. Then we describe the methodology used in designing the various components. Finally we mention the libraries developed and their usage and the current status of the system.

Figure 3 shows various components of the system. The system consists of:

An input block: comprising the *protocol essence* in C++, the *configuration information* and an optional set of Lex-Yacc rules for generating the user interface.

The tool: The tool reads the configuration information along with the Lex-Yacc rules and generates a header file, a C++ code that defines various methods to send and receive the packets of the protocol that is prototyped and interface related code.

[4]In fact, our tool uses the ACE-ASX framework as a platform to prototype application-level protocols. The library accompanying the tool provides various classes that inherit from the classes defined in the ACE-ASX framework.

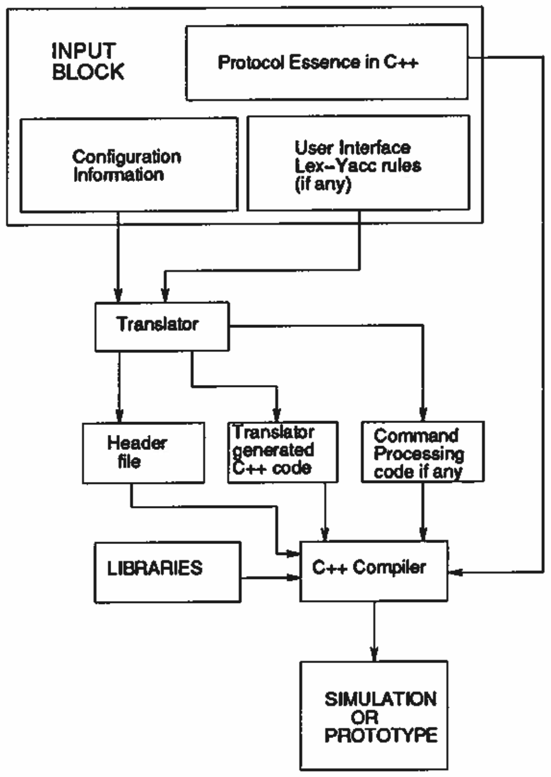
INPUT BLOCK
Protocol Essence in C++
Configuration Information
User Interface Lex-Yacc rules (if any)
Translator
Header file
Translator generated C++ code
Command Processing code if any
LIBRARIES
C++ Compiler
SIMULATION OR PROTOTYPE

Figure 3. Components of the System

The libraries: that define various base classes along with their methods. The tool-generated header file defines a protocol class that inherits from one of the classes available in the libraries.

A C++ compiler: that is essential to compile and link all the tool-generated code with the protocol essence and the libraries and to produce a simulation or a prototype of the protocol.

2.1 Design Choices

Our philosophy is to simplify protocol implementation as much as possible. This is reflected in many of our design decisions made during the development of the tool. These include:

1. **Language for specification and implementation:**

   We have chosen C++ as the language in which a protocol implementor will provide the protocol essence. C++ was chosen for a number of reasons:

   - The libraries we define inherit functionality from the classes defined in the ACE-ASX framework. ACE-ASX is developed in C++ and hence C++ was the language choice we made.
   - Object-oriented languages such as C++ offer type secure class interfaces.
   - A number of sophisticated class libraries built in C++ are available in the public domain. We use the LEDA [7] class library that provides a number of data structures and efficient algorithms in our tool implementation.

  - Sophisticated debugging tools such as *dbx* or *gdb* are readily available for C/C++ and hence the debugging task is simpler.

2. **Logical versus Wire view of packet formats:**

   The packets that are transmitted over the network often have extremely complex encoding. We do not require the implementor to provide code that references these complex packet formats. Instead, the implementor chooses some logical view of the packet and provides mapping rules which the tool uses to convert between the logical and the wire format of the packet.

3. **Using available tools for interface code generation:**

   It was observed that in many application level protocols, too much of the client software deals with user interface and command line processing. Instead of hand-crafting code that does the command line processing, we have made a choice of using *Lex* and *Yacc* utilities to generate code for the interface.

2.2 Components of the System

In this section, we describe the various components of the system.

2.2.1 The Input Block

The input block comprises two mandatory and one optional component. The mandatory components include:

1. The protocol configuration information

2. The protocol essence

The optional component comprises the Lex-Yacc rules used for generating the interface related code.

1. **Configuration Information**

   The configuration information provides information to the tool that is required to generate code on a desired platform. The configuration file consists of:

   (a) **Protocol Information**

   The keyword *configure* is used to inform the tool regarding protocol configuration specific information. This includes:

   - *name*: to indicate the name of the protocol. The name will be used by the tool as a prefix in all generated file names.
   - *platform*: to indicate the target environment, so that the tool can generate code that uses relevant libraries.
   - *channel*: to indicate the type of underlying message deliver service to be used by the protocol.

   Values for channel type can be formed by combining various pre-defined constants to obtain:

   - Connection-oriented, reliable service

- Connectionless, unreliable service

(b) **Constants and Enum declarations**

This section of the configuration file defines various C++ style constants and enumerated values that are used by the protocol essence.

(c) **Logical packet format declarations**

This section defines the logical view of the various packet formats defined by the protocol. The user uses standard C/C++ *struct* declarations to define the logical packet formats. An example of the TFTP **RRQ** packet format is shown below:

```
struct RRQPacket{
       OPCODE   opcode;
       string   filename;
       string   mode;
};
```

(d) **Wire packet format declarations and Mapping rules**

The section defining the mapping rules between the logical and the wire formats is introduced using the keyword *MAP*. The mapping rules implicitly define the wire formats for the various packet formats defined by the protocol. We use extended regular expressions to define the wire packet formats. In the TFTP example, the wire format and mapping rules for the RRQ packet are defined as shown below:

```
MAP {
    RRQPacket {
       opcode  : byte^2;
       filename: [^\0]*\0;
       mode    : [^\0]*\0;
    };
};
```

(e) **Protocol Class Declaration**

The user declares a C++ class for the protocol and declares the state variables as private variables of the class. The various protocol functions defined in the protocol essence are declared as public class methods of this protocol class. For the TFTP example, an excerpt of the class declaration is shown below:

```
class tftpclient{
    bool traceflag;
    /* others */
public:
    int  SendWRQ();
    /* others */
};
```

Appendix A provides excerpts of the configuration file for the TFTP client.

2. **Protocol Essence**

The protocol implementor supplies the protocol essence in C++ which defines protocol functions that are declared in the configuration file. An excerpt from the TFTP client protocol essence is shown below. The code illustrates the simplicity with which the client makes a RRQ request to the server. References to a packet's contents are specified logically, without mention of the positionally sensitive wire format. For example, while opcode occupies two bytes in the wire format, the logical view of an opcode is simply an `int` in C/C++. The details of an opcode's physical representation are maintained by the input and output routines, automatically generated from the wire and logical format specifications.

```
// Send a RRQ request
int tftpclient :: SendRRQ()
{
        INET_Addr       peerAddr;

        get_peer_addr(peerAddr);
        expBlock = 1;

        /* fill the RRQ packet */
        rrqPacket->opcode = RRQ;
        rrqPacket->filename = remfname;
        rrqPacket->mode = transfermode;

        /* Send the packet to the peer address */
        SendPDU(rrqPacket, peerAddr);

        // set the timer
        ev->event = DATATIMER;
        StartTimer(ev, RetransmitTimerVal);
        return 0;
}
```

Appendix B provides excerpts of the protocol essence for the TFTP client.

The tool requires the protocol essence to define the following routines:

(a) **handle packet type routine**
For every packet type supported by the protocol, a corresponding handle routine must be provided, e.g. *handle_RRQ_Packet* for the logical declaration *struct RRQ_Packet*.

(b) **handle_timeout**
The handle_timeout routine is invoked by the *timer-related* routines defined in the library whenever a preset timer expires.

(c) **Protocol specific routines**
This includes any method in the class declaration other than the handle routines.

3. **Lex-Yacc rules**

This is the optional component of the input block. It comprises Lex and Yacc rules to generate code for various interface related parts of the protocol.

2.2.2 Tool

The tool is made up of two parts:

1. **Parser**

   The parser component parses the configuration file and passes information to the code generator.

2. **Code Generator**

   The code generator is responsible for generating the header and C++ definition files according to the information provided in the configuration file. The code generator is also responsible for generating the interface related code.

2.2.3 Generated Code

The tool generates a header file that copies the protocol class declaration from the configuration file. The class from which it inherits depends on the channel type specified in the configuration file. The C++ definition file that is generated, defines additional methods of the protocol class that are not defined in the protocol essence. These methods provide an interface with the underlying operating system. The methods defined in the generated code are:

1. **handle_input method**

   The handle_input is a general purpose protocol method that is invoked by the *Reactor* mechanism provided by the ACE-ASX framework on receipt of a packet over the network. The sole purpose of this method is to decipher the type of packet received and invoke the appropriate user provided handle method for the packet type.

2. **SendPDU method**

   These methods are provided for each packet format that the protocol supports. The SendPDU methods build a wire packet from the logical packet by invoking the appropriate BuildPDU method and then invoke the appropriate operating system call for transmitting the wire packet.

3. **BuildPDU and ExtractPDU methods**

   These methods respectively build a wire packet from a logical packet and a logical packet from a wire packet using the mapping rules.

Appendix C provides excerpts from the generated code.

2.2.4 Support Libraries

The library support provided for building simulations and prototypes defines various C++ base classes from which the protocol class publicly inherits. In the current version, the library defines two base classes:

1. **TransportDGRAMNode**: that provides connectionless datagram message delivery service to the application-level protocol.

2. **TransportSTREAMNode**: that provides connection-oriented, reliable message delivery service to the application-level protocol.

The above-mentioned classes have been developed for the ACE-ASX framework on the SunOS4.1.3 and SunOS5.4 platforms and the simulation platform. Each class provides the following methods:

1. **send**: To send a packet to the remote peer entity.
2. **recv**: To receive a packet from the remote peer entity.
3. **OpenChannel**: To open a communication channel between the communicating entities.
4. **CloseChannel**: To close the already opened channel.
5. **InitServer**: This is a special method provided only for the server application. This method can provide a *concurrent* or an *iterative* server depending upon the arguments passed.

In addition, the library also provides a set of methods to *set* and *cancel* specific timers. The necessary operating system synchronization mechanisms like *poll* or *select* are built into these timer routines.

2.3 Current Status

The currently supported set of platforms include:

- **Discrete Event Simulator**: Currently, the libraries provide support for a simulator that is able to execute the protocol essence and depict its behavior. The simulator just helps to observe the protocol behavior independent of operating system interferences.
- **Application-level Network Interfaces**: These include:
    1. using ACE-ASX framework on SUNOS 4.1.3
    2. using ACE-ASX framework on SunOS 5.4

2.4 Implementation Details

We have developed all the libraries using C++. The ACE-ASX software has been used extensively. Currently we are able to generate prototypes using the ACE-ASX framework on SunOS4.1.3 and SunOS5.4 platforms as well as a simulator. The libraries used for the ACE platform make use of a number of classes supplied by the ACE library. The classes defined in our library inherit from the ACE classes. The simulator library also provides with the same interface as the libraries developed for the other platforms. The parser component of the tool has been developed using Lex and Yacc utilities.

3 Experience

This section describes our experience prototyping the TFTP and the SMTP protocol. For the TFTP protocol, a comparative analysis in terms of code size and efficiency is reported for our generated implementation of the TFTP client and the BSD client. We implemented both client and server. We report results for our generated implementation of the client. For the SMTP protocol, we provide results for the code size of the user interface and compare it with the code size of the BSD implementation.

We first present a comparison between the BSD and our implementation of the TFTP client entity in terms of code size required for some of the important tasks of the client module.

Description of part of code	**Approx. Lines of Code**	
	BSD client	**Our client**
command line processing	70	30
get/put command handling	80	30
setting a RRQ/WRQ packet	20	13
protocol essence	1193	344
Total	1363	417

We now present a comparison between the BSD and our implementation of the SMTP protocol in terms of code size required for some of the command processing tasks. For the BSD implementation's code size, we provide approximate values for the lines of code since we had to extract the SMTP code from the code for *sendmail*.

Description of part of code	**Approx. Lines of Code**	
	BSD smtp	**Our smtp**
Command interpreter	30	1 call to library routine
Decoding the received command	20	Free. Our mapping rules generate the code for us.
Parsing Address	1200 (file:parseaddr.c)	420 (lex and yacc rules)
Handling Helo command	20	10
Handling Mail command	100 (decoding, parsing)	30

Finally we present results of an experiment that compares transferring a 1.22MB file using the BSD client and our generated implementation of the client on SunOS4.1.3 and SunOS5.4 platforms. The experiment was carried out in the following setting:

- Server on SunOS4.1.3 platform: Under this setup, the server was compiled using GNU g++ version 2.7.0. This experiment was carried out in two parts:
    1. Local client compiled using GNU g++ version 2.7.0.
    2. Remote client on SunOS4.1.3 platform compiled using GNU g++ version 2.7.0.
- Server on SunOS5.4 platform: Under this setup, the server was compiled using Sun C++ version 3.0.1. This experiment was carried out in two parts:
    1. Local client compiled using Sun C++ version 3.0.1.
    2. Remote client on a SunOS4.1.3 platform compiled using GNU g++ version 2.7.0.

The BSD tftp client implementation was slightly modified. It uses a *Read-ahead/Write-behind* strategy using double buffering technique. Since our generated code does not produce code that uses such optimizations, we had to modify the tftp client code to remove the buffering strategy and use simple reads and writes. All programs were compiled with the compiler's optimization option set. For each setting, we measured the time taken for the *get* and *put* operation. All file transfers were made using the binary mode of file transfer. Each operation was performed ten times and the average time for the transfer was measured. All the time measurements are in seconds.

TFTP Client Impl	Server on SunOS4.1.3				Server on SunOS 5.4			
	Remote Client		Local Client		Remote Client		Local Client	
	get	put	get	put	get	put	get	put
BSD tftp	7.17	7.53	6.01	5.76	10.52	9.95	4.61	4.50
Our tftp	9.02	9.33	8.15	8.07	11.97	11.49	6.71	6.60

From the results, we observe that our generated implementation of the client performs almost as well as the BSD implementation; at the same time, the code size requirement for our implementation of the client is approximately 3 times less than that required for the hand-crafted version.

We obtained a profile of the execution of the BSD client and our generated client in order to assess the performance differences between the two. We show relevant parts of the profile information obtained using *GNU gprof*.

Profile for BSD tftp

```
  %  cumulative   self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 34.3      7.72     7.72    23938     0.32     0.32  _sendto [8]
 25.7     13.49     5.77    23933     0.24     0.24  _recvfrom [9]
 13.2     16.45     2.96    11957     0.25     0.25  _write [11]
 13.0     19.37     2.92    47750     0.06     0.06  _setitimer [12]
  5.1     20.52     1.15    11947     0.10     0.10  _read [13]
```

Profile for Our tftp

```
  %  cumulative   self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 24.0     10.36    10.36    26280     0.39     0.39  _sendto [2]
  9.5     14.45     4.09    52526     0.08     0.08  _select [7]
  9.1     18.37     3.92    14351     0.27     0.27  _write [9]
  8.2     21.91     3.54    26274     0.13     0.13  _recvfrom [10]
  6.0     24.49     2.58   131347     0.02     0.02  _gettimeofday [13]
  4.3     26.35     1.86                             _char2byte__6BUFFERRci
[16]
  4.3     28.20     1.85    11948     0.15     0.15  _read [17]
  3.8     29.85     1.65                             _byte2char__6BUFFERiRc
[18]
  3.7     31.45     1.61    26262     0.06     0.06 _ExtractPDU__10DataPacketP
6BUFFER [19]
```

Our implementation of the client uses an additional system call *select* which is not used by the BSD implementation. The select system call is used by the underlying ACE-ASX framework in its *Event handling* mechanism which invokes routines such as *handle_input* when data is received at the socket. Additional overhead in our client is caused by copying of data from the received packet into the logical structure (byte2char) and copying data from logical structure into the physical packet (char2byte). If this is a problem, we can replace the extra copy by compiling the user calls to read and write the logical packet by macro calls that read and write the physical packet.

Our automatically generated client and server did not incorporate some of the smart strategies used by the hand-crafted BSD implementation such as *Read-Ahead/Write-Behind* using *double buffering*. There are two approaches to incorporating such optimizations into network software:

1. The client essence itself could be augmented to include such optimizations. This would imply that the user has to provide additional hand-crafted code for such optimization schemes to work.

2. The tool could be improved to automatically insert such optimizations where appropriate.

Because our ultimate goal is a tool whose optimization rivals hand-crafted network software, we would derive no benefit from including the above optimizations in the client protocol essence. We plan instead to improve our tool, including even more aggressive optimizations as discussed in Section 4.

3.1 Preliminary Results on the use of Powerful Set of Regular Expressions

In the current implementation of the tool, we do not support a wide range of regular expressions for the mapping rules. Experience with the SMTP protocol indicated a necessity for support for powerful regular expressions in the mapping rules.

The SMTP *MAIL* or *RCPT* command consists of a *path* argument which contains optional relay host addresses and a mailbox of the destination user. A typical command may look like:

```
MAIL FROM:<@cs1,@cs2.school.edu:user@host.addr>
```

In the current implementation, owing to the lack of support for powerful regular expressions, we had to declare the path as a character string in the logical definition as shown:

```
struct MAIL_PDU{
        string   code;
        string   path;
};
```

On receipt of the packet, the protocol essence is required to decode the path and separate the various parts that make up the path. This task can be significantly reduced if we could specify the logical structure of the packet and the mapping rule as shown:

```
struct PATH{
        string  start;
        string  atdomain;
        string  user;
        string  domain;
        string  end;
};

struct MAIL{
        string code;
        string ws;
        string from;
        PATH    rpath;
        string  terminate;
};


MAP{
   MAIL{
       code : "MAIL"
       ws   : " "+
       from : "FROM:"
```

```
        rpath : (
                      start : "<"
                      atdomain :
                         ((@[a-zA-Z]+([.][a-zA-Z]+)*,)*@[a-zA-Z]+([.][a-zA-Z]+)*:)?
                      user : [a-zA-Z]+
                      domain : @[a-zA-Z]+([.][a-zA-Z]+)*
                      end    : ">"
                )
    }
}
```

Thus the generated code will handle the decoding task and relieve the user from supplying code for this task.

In the current experimental version, we have hand-crafted code that would otherwise be generated by the tool, that does the decoding based on the powerful regular expressions. In this regard we have used the *GNU regex* library to handle the regular expressions and match the appropriate strings in the incoming packet.

4 Conclusions and Future Work

In this paper we have demonstrated the usefulness of a tool that can read simple C/C++ language implementations of protocols and generate code for a simulator and a prototype on a wide variety of platforms without any modifications to the input. We have also demonstrated the effectiveness of the user viewing a packet format at a logical level and the tool converting it into the appropriate wire format. The libraries provided by the tool eliminate any need for the protocol essence to worry about lower-level operating system details, especially detection of various external events and synchronization issues. The essence is only required to provide code to handle such events once they have been detected. Finally, the Lex and Yacc based rules simplify the user-interface code generation significantly.

While our techniques could be applied to other platforms such as OS kernels and hardware, we have concentrated on application protocols. We have also concentrated on protocols for which rapid prototyping and ease of maintenance are more important than fine-tuning of performance. We believe this is a good choice because there are a large number of application protocols that are and will be deployed, and for many of them (e.g., management protocols) functionality is more more important than performance. We note that our tool does provide reasonably good performance; however, fine-tuned performance will require automatic optimization techniques that will be part of the second stage of this project.

Our current research plans call for improving the basic tool along the following lines:

- **Add more target platforms:** In the short term, we want to add support for TLI (transport layer interface) on the Solaris platform, and improve our simulator platform. In the long term, we want to consider a kernel environment and possibly stand-alone hardware platforms.
- **Increase flexibility of tool:** There are a number of aspects of our tool that we can generalize. These become more apparent as we implement more examples. For instance, we want to allow a powerful set of regular expressions[5] to be specified in the mapping rules. We also want to provide support for *little endian* machines.

[5]We are already working on a scheme that makes use of the GNU *regex* library.

- **Adding Fault-tolerance Automatically:** Whenever a protocol attempts to use operating system resources such as buffers or sockets, the protocol must deal with the case that the resource is unavailable; the implementor must write what we call *platform-specific fault-tolerance code*. Can this code be generated automatically? In simple cases, where the protocol is implemented by a process that exits, this is very easy. More sophisticated schemes that involve retrying and automatically keeping track of used resources (in order to free them) are also possible.

The second stage of our project, in which we automatically add optimizations, has not begun in earnest. We have the following research ideas and plans:

- Make basic tool operations more efficient. For instance, we plan to remove the extra copy caused by logical to physical conversions by replacing user calls to the logical packet with Macros that reference the physical packet. We also plan to inline, as far as possible, the methods defined by the various classes in the libraries.

- Automate existing paradigms (e.g., header prediction, caching, using hints) that have been hand-crafted into existing protocol implementations. In particular, we have a strategy for automating header prediction, in which tests for the expected case are optimized.

- Automatically customize common protocol functions like lookups to the reference patterns of particular environments [5].

- Extend existing work in compiler optimizations [3, 1] applied to network protocol implementations.

- Apply existing techniques from the work on optimizing compilers to unify various ad hoc techniques.

Our vision is that the writing of network software should become an engineering discipline, one that can be easily taught to and mastered by undergraduates. Before the advent of compilers and other software tools, even ordinary programming was the domain of a few abstruse wizards. We hope that approaches similar to our tool may pave the way for similar transformation in the area of network programming.

References

[1] M. Abbott and L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *ACM Transactions on Networking*, 1(5), October 1993.

[2] D. Bacon, A. Dupuy, J. Schwartz, and Y. Yemini. Nest: A Network Simulation and Prototyping Tool. In *Proceedings of the Winter 1988 Usenix Conference*, Dallas, 1988. USENIX.

[3] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, Philadelphia, Pennsylvania, September 1990. ACM.

[4] N. C. Hutchinson and L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1), January 1991.

[5] Raj Jain. Packet Trains: Measurements and a New Model for Computer Traffic. *IEEE Journal on Selected Areas in Communications*, 4(6):1162–1167, May 1986.

[6] SUN Microsystems. XDR: External Data Representation Standard. Technical Report RFC 1014, Sun Microsystems, Inc, June 1987.

[7] Stefan Naher, Kurt Mehlhorn, and Christian Uhrig. *The LEDA User Manual Version R3.2*, July 1995.

[8] Sean O'Malley, Todd Proebsting, and Allen Brady Montz. USC: A Universal Stub Generator. In *SIGCOMM 94*, London, UK, August 1994. SIGS.

[9] J. Postel. SIMPLE MAIL TRANSFER PROTOCOL. Technical Report RFC 821, Information Sciences Institute, August 1982.

[10] Douglas C. Schmidt. IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services. *C++ Report*, November/December 1992.

[11] Douglas C. Schmidt. The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications. In *Proceedings of the* 11th *Annual Sun Users Group Conference*, San Jose, CA, December 1993. SUG.

[12] Douglas C. Schmidt. ASX: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the* 6th *USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994. USENIX.

[13] R. Sijelmassi and B. Strausser. The Distributed Implementation Generator: an overview and user guide. NCSL/SNA Technical Report 91-3, National Institute of Standards and Technology, January 1991.

[14] R. Sijelmassi and B. Strausser. The Portable Estelle Translator: an overview and user guide. NCSL/SNA Technical Report 91-2, National Institute of Standards and Technology, January 1991.

[15] K. Sollins. The TFTP Protocol (Revision 2). Technical Report RFC 1350, Massachusetts Institute of Technology, July 1992.

A TFTP Client Configuration File

```
configure {
        name    =       tftp;
        suffix  =       client;
        mtu     =       MAXPDUSIZE;
        platform=       ACE;
        channel =       UNRELIABLE |  CONNECTIONLESS;
}

/*      define all sorts of constants */
const   int MAXPDUSIZE   =      516;
const   int MAXDATASIZE  =      512;
const   int SERVERPORT   =      3000;

/* define the values for opcodes */
typedef enum {
        RRQ = 1,
        WRQ = 2,
        DATA = 3,
        ACK = 4,
        ERROR = 5
} OPCODE;

/* define timer types */
typedef enum {
        DATATIMER,
        ACKTIMER
} TIMER;

/* define logical packet views of various packet types of TFTP */
struct RRQPacket{
        OPCODE  opcode;
        string  filename;
        string  mode;
};

struct WRQPacket{
        OPCODE  opcode;
        string  filename;
        string  mode;
};

struct DataPacket{
        OPCODE  opcode;
        int     block;
        int     len;
        char    data[MAXDATASIZE];
};

struct AckPacket{
        OPCODE  opcode;
        int     block;
};

struct ErrorPacket{
        OPCODE  opcode;
        int     block;
        string  errMsg;
};

/* define the mapping rules between LOGICAL <==> WIRE */
map {
    RRQPacket{
        opcode   :      byte^2 { IFF(opcode == RRQ) }
        filename :      [^\0]*\0;
        mode     :      [^\0]*\0;
    };
    WRQPacket{
        opcode   :       byte^2 { IFF(opcode == WRQ) }
        filename :       [^\0]*\0;
        mode     :       [^\0]*\0;
    };
    DataPacket{
        opcode   :       byte^2 { IFF(opcode == DATA) }
        block    :       byte^2;
        len      :       byte^0 { len = PDU_LEN() - FIELDLEN(opcode) -
                                  FIELDLEN(block); }
        data     :       (byte^1)^len;
    };
    AckPacket{
        opcode   :       byte^2 { IFF(opcode == ACK) }
        block    :       byte^2;
    };
    ErrorPacket{
        opcode   :       byte^2 { IFF(opcode == ERROR) }
        block    :       byte^2;
        errMsg   :       [^\0]*\0;
    };
};

/* define the TFTP protocol class */
class tftpclient {
        int             sockfd;
        short           peerport;
        string          peerhost;
        short           fd;             // local file desc
        string          remfname;       // remote filename
        string          transfermode = "netascii";
        int             RetransmitTimerVal = 5;
        int             MaxTimeoutVal = 25;
        short           expBlock;
        bool            traceflag = false;
        bool            verboseflag = false;
        short           connected = 0;
public:
        void    command(int);
        void    init();
        void    put(char *, char *, char *);
        void    get(char *, char *, char *);
        void    connect(char *, char *);
        void    quit();
        int     handle_user_input(void *);
        int     handle_timeout(const Time_Value &tv, const void *);
        int     handle_DataPacket(DataPacket *, const INET_Addr &);
        int     handle_AckPacket(AckPacket *, const INET_Addr &);
        int     handle_ErrorPacket(ErrorPacket *, const INET_Addr &);
        int     handle_DATA_timeout();
        int     handle_ACK_timeout();
        int     SendRRQ();
        int     SendWRQ();
};
```

B TFTP Client Protocol Essence

```
// Command handling routine invoked by interface related code
int tftpclient :: handle_user_input (void *param)
{
    CMD_PARAMS  *cp = (CMD_PARAMS *)param;

    switch (cp->type){
        case GET:       get(cp->s1, cp->s2, cp->s3);
                        break;
        case PUT:       put(cp->s1, cp->s2, cp->s3);
                        break;
        case CONNECT:   connect(cp->s1, cp->s2);
                        break;
        case QUIT:      quit();
                        break;
        default:
                        break;
    }
}

// Send a RRQ request
int tftpclient :: SendRRQ()
{
        INET_Addr       peerAddr;

        get_peer_addr(peerAddr);
        expBlock = 1;
        rrqPacket->opcode = RRQ;
        rrqPacket->filename = remfname;
        rrqPacket->mode = transfermode;
        StartClock();   // for statistics
        SendPDU(rrqPacket, peerAddr);
        ev->event = DATATIMER;
        StartTimer(ev, RetransmitTimerVal);
        return 0;
}

// Send a WRQ request
int tftpclient :: SendWRQ()
{
        INET_Addr       peerAddr;

        get_peer_addr(peerAddr);
        expBlock = 0;
        wrqPacket->opcode = WRQ;
        wrqPacket->filename = remfname;
        wrqPacket->mode = transfermode;
        StartClock();   // for statistics
        SendPDU(wrqPacket, peerAddr);
        ev->event = ACKTIMER;
        StartTimer(ev, RetransmitTimerVal);
        return 0;
}

// Handle a DATA Packet
int tftpclient :: handle_DATA(DataPacket *packet, INET_Addr &tempAddr)
{
   CancelTimer(ev);
   // if it is a first packet, set the address of the peer as the newly
   // received address.
   if (firstPacket) {
        firstPacket = false;
        SetPeerEntry(tempAddr);
   } else {
           if (!MatchWithPeer(tempAddr)){
                // send error message to wrong peer
                perror("Entries don't match");
                return -1;
            }
    }
    if (lastPacket){     // We have received the last packet again. So we
                         // Dally and send the same ACK again
        if (verboseflag)
                cout << "CLIENT: sending dally ack #" << ackPacket->block << endl;
        SendPDU(ackPacket, tempAddr);
        cout << "client:SUCCESS\n";
        return 0;
    } else if (packet->block == expBlock){
        timeout = 0;
        ::write(fd, packet->data, packet->len);
        totalData += packet->len;
        // Build an ACK packet
        ackPacket->opcode = ACK;
        ackPacket->block = expBlock;
        if (verboseflag)
                cout << "CLIENT: sending ack #" << ackPacket->block << endl;
        SendPDU(ackPacket, tempAddr);
        expBlock++;
        if (packet->len < MAXDATASIZE){
                lastPacket = true;
                return 0;
        }
        ev->event = DATATIMER;
        StartTimer(ev, RetransmitTimerVal);
        return 0;
    }
    // invalid packet block
    cout << "Client :Invalid Block # " << packet->block << " Expected "
                << expBlock << endl;
    //Synchronize both sides
    int packetsFlushed = SynchNet(MAXPDUSIZE);
    if (verboseflag)
        cout << "Total packets flushed : " << packetsFlushed << endl;

    if (packet->block == expBlock-1) // other side is one behind us
        return handle_DATA_timeout();
    return 0;
}

// Handle timeout for DATA Packet
int tftpclient :: handle_DATA_timeout()
{
        INET_Addr       peerAddr;
        get_peer_addr(peerAddr);
        if (firstPacket){       // no response for our RRQ packet
                perror("handle_DATA_timeout: No response from peer");
                return -1;
        } else {
           // send Same ACK again
        if (verboseflag)
                cout << "Client:sending Ack# " << ackPacket->block << " again\n";
           SendPDU(ackPacket, peerAddr);
           ev->event = DATATIMER;
           StartTimer(ev, RetransmitTimerVal);
           return 0;
        }
}
```

C Excerpts of the Generated C++ code for sending/receiving

```
#include "tftpserver.h"

RRQPacket       *__RRQPacket = new RRQPacket;
WRQPacket       *__WRQPacket = new WRQPacket;
ErrorPacket     *__ErrorPacket = new ErrorPacket;
DataPacket      *__DataPacket = new DataPacket;
AckPacket       *__AckPacket = new AckPacket;
BUFFER *sendBuffer = new BUFFER(MAXPDUSIZE);
BUFFER *recvBuffer = new BUFFER;
unsigned char *netBuff = new unsigned char[MAXPDUSIZE];

void    RRQPacket :: BuildPDU(BUFFER *buffer)
{
        int     i,j;
        _pduLen = buffer->ByteLength();
        _size_opcode = buffer->enum2byte((unsigned short )opcode,2);
        _size_filename = buffer->string2bytestream(filename);
        _size_mode = buffer->string2bytestream(mode);
        _pduLen = buffer->ByteLength() - _pduLen;
}
bool    RRQPacket :: ExtractPDU(BUFFER *buffer)
{
        int     i,j;
        unsigned short __us = 0;
        _size_opcode = buffer->byte2enum(2, __us);
        opcode = (OPCODE)__us;
        IFF(opcode == RRQ)
        _size_filename = buffer->bytestream2string(filename);
        _size_mode = buffer->bytestream2string(mode);
        return true; // SUCCESSful extraction
}

int     tftpserver :: SendPDU(RRQPacket* packet, const INET_Addr &addr)
{
        packet->BuildPDU(sendBuffer);
#ifdef DEBUG
        sendBuffer->DumpBuffer();
#endif
        int status = TransportDGRAMNode::send(sendBuffer->GetBuf(), packet->_pduLen, addr);
        sendBuffer->ClearBuf();
        return status;
}
int     tftpserver :: SendPDU(RRQPacket* packet)
{
        packet->BuildPDU(sendBuffer);
#ifdef DEBUG
        sendBuffer->DumpBuffer();
#endif
        int status = TransportDGRAMNode::send(sendBuffer->GetBuf(), packet->_pduLen);
        sendBuffer->ClearBuf();
        return status;
}

int     tftpserver :: handle_input(int _fd)
{
        int _len;
        INET_Addr addr;
        bool __first_time = true;
        int stat;
        BUFFER  *buffer;

        do{//A do loop is used because we know for sure that
                // a call to handle_input means something has arrived
                ::memset(netBuff,0,MAXPDUSIZE);
                _len = TransportDGRAMNode::recv((void *)netBuff, MAXPDUSIZE, addr);
                if (_len <= 0) break;
                recvBuffer->SetBuff(_len, netBuff);
                stat = decipherPDU(recvBuffer, addr);
                if (stat == -1){
                        return -1;
                }
        }while(this->MoreInput(_fd));
        return 1;
}
int     tftpserver :: decipherPDU(BUFFER *buffer, INET_Addr &addr)
{
#ifdef DEBUG
        buffer->DumpBuffer();
        buffer->ResetBuf();
#endif
        // now go thru all packet types and find which one this is
        {
                if (__RRQPacket->ExtractPDU(buffer)){
                        __RRQPacket->_pduLen = buffer->GetPDULength();
                        return handle_RRQPacket(__RRQPacket, addr);
                }
                // Not matched
                buffer->ResetBuf();
        }
        {

                if (__WRQPacket->ExtractPDU(buffer)){
                        __WRQPacket->_pduLen = buffer->GetPDULength();
                        return handle_WRQPacket(__WRQPacket, addr);
                }
                // Not matched
                buffer->ResetBuf();
        }
        {

                if (__ErrorPacket->ExtractPDU(buffer)){
                        __ErrorPacket->_pduLen = buffer->GetPDULength();
                        return handle_ErrorPacket(__ErrorPacket, addr);
                }
                // Not matched
                buffer->ResetBuf();
        }
        {

                if (__DataPacket->ExtractPDU(buffer)){
                        __DataPacket->_pduLen = buffer->GetPDULength();
                        return handle_DataPacket(__DataPacket, addr);
                }
                // Not matched
                buffer->ResetBuf();
        }
        {

                if (__AckPacket->ExtractPDU(buffer)){
                        __AckPacket->_pduLen = buffer->GetPDULength();
                        return handle_AckPacket(__AckPacket, addr);
                }
                // Not matched
                buffer->ResetBuf();
        }
#endif
        // not matched with any packet
        cerr << "handle_input unable to match with any packet\n";
        return -1;
}
```