A Protocol Language Approach to Generating Client-Server Software

Melvin A.L. Douglas and Philip K. Chan*
Department of Computer Sciences
Florida Institute of Technology
Melbourne, FL 32901,USA
Email: Melvin_Douglas@yahoo.com & pkc@cs.fit.edu
Contact author phone: 321-674-7280, fax: 321-674-7046

Technical Report: CS-2000-2

Abstract

Client-server software is becoming more common as the Internet grows. To ease the burden of repeatedly writing low-level communication and protocol code, we seek to design a protocol language, "*My Simple Protocol Language*" *(MSPL)*, that produces the corresponding communication functions. The programmer then supplies the rest of the application-specific code but never modifies the generated code. Besides saving development time, this approach also reduces programming errors. The potential to develop more efficient code also exists once the technique of generating code is mastered. The main contribution, however, is that unlike RPC, Corba or RMI, we provide the user with not only functions that take care of lower level communication data structures, but also the ordering and format of messages (protocol) which are specified in *MSPL* programs. The *MSPL* programs are then passed to the *MSPL Compiler*, which produces the low-level communication and protocol modules. These protocol modules are then linked to other user-written modules to produce the final software application.

1. Introduction

There are a number of advantages that arise if the protocol language developed during this research is used for production of quality code. Not least of these is the potential for reducing the risk and cost of software development, by reducing the potential for the introduction of errors, and increasing the speed with which software can be produced. The magnitude of these advantages is increased where the risk and

cost of software production is higher, such as in the case of high-integrity systems development. In order to derive these benefits, it is vitally important to ensure that the generated code is functionally faithful to its specification. Current methods work relatively well but they use high-level languages, which are not geared towards developing communication protocols. This leads to code developed by programmers that is not robust or very efficient. It is usually very hard to read and therefore, almost impossible to maintain.

There are two problems that we focus on and provide a solution to in this paper. The first is providing a protocol language that is capable of solving the problem of writing client-server software efficiently and reliably. The protocol language allows the specification of application-level client-server protocols. The second task is to demonstrate the feasibility of using the protocol language developed on '*real world*' protocols like HTTP RFC 2616.

Section 2 gives an extended overview of related work. Techniques used in this area of research in the past are discussed and compared. Section 3 looks more closely at the solutions to the problems being focused on in this paper. It explains different concepts used during research and development. An example of how *MSPL* may be used for the implementation of a user's protocol is discussed and analyzed. Section 4 analyzes the use of *MSPL* to develop clients and servers that can interact with existing servers and clients that meet standard RFC protocol specifications. Section 5 discusses the conclusions made after in-depth research, implementation and testing of the generated code.

2. Related Work

Program generation, more formally known as *software synthesis*, deals with the automation of program writing. Tools that generate programs or code are often seen as a part of a *Problem Solving Environment (PSE)*. These tools implement some kind of command or specification language. Distributed systems must communicate. Communication requires protocols to be built preferably with a manageable complexity. To communicate well requires protocols to be efficient in design and implementation. Complexity within protocols can be managed with simple interfaces that allow the protocols to be composed in a modular manner. To provide higher level functionality than is provided by

any single protocol, they are frequently composed together into protocol stacks. Each layer in the stack is linked to the layer immediately above and the layer immediately below it. The *CTADEL* system [Engelen et al., 1996] generates code for a meteorological model, which was compared with efficient hand-written production code. The authors point out that the highest efficiency of code can only be achieved by exploiting specific characteristics of computer architectures. Efficiency and portability are generally conflicting goals.

2.1 Protocol Specification Languages

Specifying protocol using a formal language has been valuable in verifying the correctness of network protocols and generating test cases to evaluate implementations. One such language is Estelle [Amer et al., 1997], which specifies protocols as a set of finite state machines. Though Estelle eases the formal verification process by machines, it is not easy to read by humans. In addition, it was not originally designed to facilitate the automated process of implementing the specified protocols. Languages like Prolac [Kohler et al., 1999], on the other hand, are designed to generate code that implements the specified protocols. These languages are easier to read by humans, but are not designed for formal verification by machines. Languages for protocol specification have been mostly focusing on the Transport, Network, and Data Link layers in the OSI model because these network protocols constitute the vital infrastructure of the Internet. However, languages for specifying protocols above the Transport layer are not well investigated, even though a large number of distributed and network applications, particularly in the client-server model, have been developed and are used in critical operations. Our proposed language, *MSPL*, allows the specification of protocols between a client and server at the Application layer. The language is not designed for formal verification; rather, it focuses on readability by humans (hence ease in programming) and generating communication code to implement the specified protocols.

2.2 BEA Tuxedo®

BEA Tuxedo supports four distinct communication methods that are versatile and easy to use yet powerful enough to build a wide variety of mission-critical business applications [BEA, 1996]. Tuxedo

and *MSPL* have the same basic goal, which is to generate client-server software from a high-level of abstraction. *MSPL* is a specification-based language that describes the protocols and generates the necessary communication modules and interface. The four communication methods supported by Tuxedo are Events-One Way, Request/Response, Conversational Interactions and Queued Communications. The user is allowed to choose one of these communication methods and then call the appropriate library-based functions. This discloses one of the main differences between *MSPL* and Tuxedo. Tuxedo supports multiple types of *send* and *receive* commands while *MSPL* supports complete specification protocols. *MSPL* allows the application programmer to focus more on the specification of the protocol while in Tuxedo, the application programmer concentrates more on actual coding and function calls. Tuxedo has much more functionality than *MSPL* currently does but *MSPL* can be extended to incorporate most of the features. The general structure of the message sent between the client and server, are very similar, almost identical. Overall, BEA Tuxedo and *MSPL* are similar in several ways but they have a fundamental difference that separates them. BEA Tuxedo supplies the application programmer with functions they can call while *MSPL* allows the application programmer to implement a complete protocol.

2.3 Sun's XDR/RPC

RPC is a simple form of the request/response paradigm, modeled after the local procedure call structure. Any remote procedure call, however, may not be able to contact the server, and thus, fail. This makes the report error types such as time-outs, very important. Many RPC systems are designed for use with the *exception handling* available in *Ada*, *Java* and many other programming languages. If the language does not have any *exception handling* capabilities, then the RPC systems usually resort to using the methods in UNIX and other conventional operating systems. The systems usually deliver a well-known value to indicate failure. This method, however, has the disadvantage of having the caller test every return value. In *MSPL* the return value or message is *tested* within the language. Although RPC is an applicable programming mechanism, it only allows procedures to be generated not the actual protocol like *MSPL* does. Similar to BEA Tuxedo, if the application programmer wanted or needed to change the

programming language, it would be necessary to re-implement the same protocol in the new *target-language*.

2.4 Library-Based and Specification-Based Approaches

Developing client-server software at a higher level of abstraction can be characterized into two main approaches. The first approach is library-based like BEA Tuxedo. The second is a specification-language approach like *MSPL*.

Library-based methods provide a fixed list of routines. Consider the choice of using *Java* or *C* versus developing a set of *Assembly Libraries*. It is possible to write software without the existence of *Java* or *C*, however, these languages provide a programmer with a higher level of abstraction for programming. Similarly, *MSPL* provides a higher level abstraction for implementing protocols (than library-based methods). This has several advantages and disadvantages.

One advantage is the fact that high-level programming languages are easier to read and understand. Another advantage is the fact they shorten the development time required. Also, by separating the specification from the implementation, the programmer only has to specify *what* to do and not *how* to do it, a key difference in declarative and procedural characterization of expressing solutions in programming languages. In addition, this separation enhances portability. On the contrary, the library-based approach is *target-language* specific, which means that if the programmer would like to change the *target-language*, the protocol code would have to be re-written in the new *target-language* because the protocol-specification is closely intertwined with the protocol-implementation.

Another major advantage of the specification-based approach is that protocols are automatically "aligned". By alignment, we mean that if the number or order of parameters for a particular request is changed on the client side, then the server side is automatically adjusted to align with these changes. In the library-based approach, changes in the server protocol module require the corresponding changes to be made in the client protocol module (or vice versa) manually. Errors on the part of the programmer, may lead to unaligned changes which will inevitably prolong development time. On the contrary, the

specification-based approach uses a compiler to generate the client and server protocol modules, which are automatically aligned. Consequently, the possibility of programmer errors is reduced and reliability in the resulting client-server software is enhanced.

However, a disadvantage of the specification-based approach is more abstraction generally leads to less control and flexibility. There is also the added responsibility of mastering another language.

3. MSPL

The first problem stated in Section 1, is handled by building *'My Simple Protocol Language' (MSPL)*, which is used to write programs that implement the communication protocol stack shown below in Figure 3 - 1.

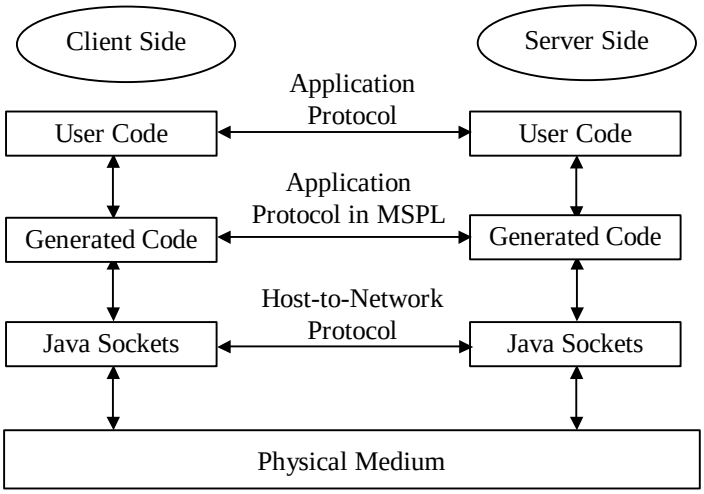

Figure 3 – 1. Client-Server Code Generation Model

The solid lines represent the actual paths of communication while the dotted lines represent the virtual communication paths. Each layer on the client side communicates with the corresponding layer on the server side. Each layer has a distinct function. The application programmer is responsible for defining the Application Protocol. This research looks at developing *MSPL* to specify an application being developed and outputted by a *MSPL Compiler* with the input of a *MSPL* program.

3.1 Architecture

Figure 3 – 2 shows the architecture of the entire client-server code generation process. First, a program representing the Application Protocol in *MSPL* must be written by the application programmer. Then it is sent to the *MSPL Compiler*, which outputs the Client Protocol Module and Server Protocol Module.

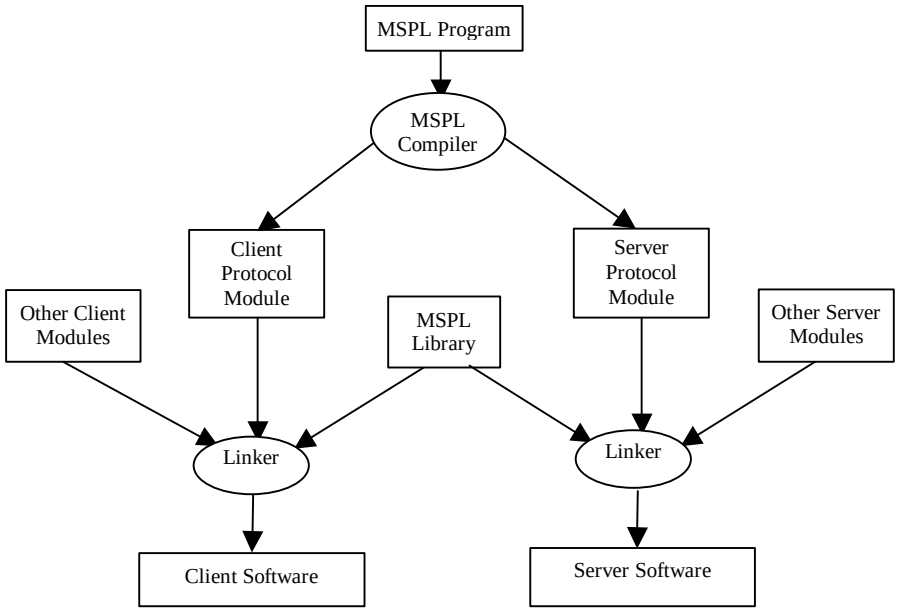

Figure 3 – 2. MSPL Architecture

These protocol modules are then linked to the MSPL Library and other user-written modules. This produces the final product of a client-server software application.

3.2 ESFTP

Before we take a closer look at exactly how to write a program in *MSPL* and how the client and server code is generated, we describe a simple protocol called the *Extremely Simple File Transfer Protocol (ESFTP)*, which will be used as a running example throughout this section. It is not the RFC 959 Standard FTP protocol. All communication takes place over one connection and the client begins the conversation instead of the server. *ESFTP* can be used to transfer files from a client machine to another machine running the server and also files from the machine running the server to any machine that has the client. *ESFTP* allows three commands: *put* a file, *get* a file and `quit` the application, thus,

closing the network connection. The protocol is also used to send error messages between the client and server.

There are a few steps of initialization that must take place before the client or server can acknowledge any of the three commands mentioned above. In the initialization phase, the server must:

1. Be started with a port number known by all clients that may request connection.
2. Open a socket and if the port is in use then print an error message and exit.
3. Listen for client connections on the specified port.

The client also has three initialization steps, which are:

1. Invoke with server address and port number.
2. Make a socket connection to the server.
3. Prompt user for requests that need to be sent to the server.

Once these initialization steps have been taken, then any of the three commands may be used. The structure and ordering of each command is given in Figure 3 – 3 below.

```
Put <filename>
    1.  Client ensures filename exists.
    2.  Client sends token Put, the filename and an integer representing the size of the file.
    3.  While the entire file has not been copied to the server:
            a.  Client sends up to buffersize bytes (where buffersize is an integer).
            b.  Server reads the bytes sent by the client.
    4.  Server sends reply message stating whether the request was completed successfully.
    5.  Client prints status message to inform the user and waits for next command/request.

Get <filename>
    1.  Client sends command token Get followed by the name of the file being requested.
    2.  Server checks and ensures the filename exists.
    3.  Server sends message stating whether file exists and the size of the file if it
        exists.
    4.   While the entire file has not been sent to the client:
            a.  Server sends up to buffersize bytes.
            b.  Client reads the bytes sent by the server.
    5.  Client informs user whether or not expected bytes are equal to actual bytes received.
    6.  Client and server wait for next command/request from user.

Quit
    1.  Client sends command token Quit and then closes the connection
    2.  Server receives command and also closes its end of the connection.
```

Figure 3 – 3. ESFTP Command Specification

3.3 Implementing ESFTP in MSPL

In this section, we take a closer look at exactly how to write a program in *MSPL* and how the client and server code is generated. In Figure 3 – 4 the specification-protocol is written for *ESFTP* (this is the same program that generates the portions of code shown in Figure 3 – 6).

```
1.  # MSPL file used to generate code for the ESFTP Application
2.  Parameters
3.     defaultClientPort 55000,  # between 0 and 65535
4.     defaultServerPort 55000,  # between 0 and 65535
5.     bufferSize 1000, #same size buffer for Client and Server
6.     maxClientsSupported 9;
7.  Begin
8.     Request Get # method for client to receive a file from server
9.        Constant String "Get ",
10.       String Filename,
11.       Constant String "\r\n";
12.    Reply Ok
13.       Constant String "200 request executed successfully \r\n",
14.       int statusref,
15.       int length,
16.       byte[] actualFile;
17.    Reply noFile
18.       Constant String "400 file does dot exist \r\n";
19.    Request Put       # method for client to send a file to the server
20.       Constant String "Put ",
21.       String Filename,
22.       int length,
23.       byte[] actualFile;
24.    Reply Successfull
25.       Constant String "200 File transfer successful \r\n";
26.    Reply fileExists
27.       Constant String "300 File already exists \r\n";
28. End
```

Figure 3 – 4. MSPL Code For ESFTP

Every client module generated from the *MSPL* program contains a method called *`connectTo`*, which takes a string as its parameter. The method is used to establish a connection to the server. The string parameter is the *hostname* or *IP address* of where to try and connect. The server you want to connect to must already be running at that address and listening on the port specified in the *MSPL* program.

All the parameters have default values, which can be overridden. This frees the programmer from being forced to declare all of them. In this case, four parameters have been defined. Both `defaultClientPort` and `defaultServerPort` have been assigned the value of 55000 on lines 3 and 4. The `buffersize` designates the maximum size of the packets being sent between the two machines and has been assigned a value of 1000 bytes on line 5. The last parameter assigned a value is on line 6. This is the maximum number of clients that can connect to the server at any given time. All of these

parameters are defined more specifically later in Section 3.3.1 on Definable Communication Parameters. Comments may be inserted by preceding the text with a number # sign.

Line 7 signals the beginning of the Request–Reply structure. No parameters can be assigned a value after this keyword. The `get` request on line 8 sends a string value from the client to the server. The expected reply from the server is either `Ok` or `noFile` as shown on lines 12 and 17. The first token sent back in all protocols including RFC, is the name of the Reply. In this case the first token will either be `200` or `400`. The signifcance of these numbers are discussed later and described further in Figure 4 – 3. If the reply is `Ok`, then the next data type expected is an integer followed by another integer and then finally bytes. The first integer is used by the user-written modules to see if this is just a continuation of receiving a file or is it the start of receiving a new file. The second integer is the size of the file being sent and is used to inform the client of just how many bytes will be sent. Finally, the actual file is transferred in chunks no larger than the `buffersize` until the entire file has been transferred. If the reply is `nofile`, then as line 17 shows, a string follows which may contain more information as to exactly why the request was unsuccessful.

The next possible request is `put`, which is shown on line 19. This request sends the request name `put`, followed by a string for the name of the file to be sent to the server, an integer representing the size of the file to be sent and then finally bytes equivalent to or smaller than the specified `buffersize`. All these fields in the message packet are defined on lines 20, 21 and 22. The two possible replies to this request are *`Successful`* or *`fileExists`*. *`Successful`* is the name of the reply on line 23 and has a string sent back describing the current state. This simply means if the request was executed successfully then that is all the information that needs to be reported to the client. The second reply on line 25 is *`fileExists`* and is followed by a *`String`* type, which may be used to describe what the server side plans to do since the file already exists.

The last request that is present in all the generated protocol modules is the `quit` request. This request sends `quit` as a string to notify the server the connection is being closed. There are not any reply

parameters for the `quit` request. The `quit` command is not written in Figure 3 - 4 because it is standard in most protocols, therefore it is automatically generated. It can be overridden but in the case of this protocol it is not necessary. Due to space limitations, the EBNF definition of *MSPL* may be found in [Douglas, 2000].

3.3.1 Definable Communication Parameters

In Figure 3 - 4, the section between the keywords `Parameters` and `Begin`, Lines 2 to 6, is where parameters are initialized. One parameter gives the programmer control over which port to communicate. It is left up to the programmer to ensure this port is available. If the chosen port is not available, then the generated code will simply print a message saying the port is already in use, upon which, it will halt all attempts to use the port. Another parameter allows the programmer to define the buffer size in bytes for each message sent to and from the client. The blocks of data sent are guaranteed to be no larger than this number provided. The `maximumClientsSupported` parameter allows one to specify how many clients are allowed to connect to the generated server at any given moment. All the parameters have default values if the programmer does not want to specify them.

After setting all the desired parameters, the main body of code between the keywords `Begin` and `End` may be written. There is an option to send a `Handshake` which allows the server to send a message before the client does. After examining several existing protocols, it was discovered that not all client server protocols start with a request from the client side. In some instances, the server first sends a message stating it is ready to provide a service and it is running a certain version of the application. The server does not expect a reply to this message. Therefore, it is really not correct to call it a request. It simply informs the client side of some information, which is why it was named `Handshake` in *MSPL*. It is referred to as Events-One Way in Tuxedo [BEA, 1996].

3.3.2 Structure of Request–Reply Statement

Whether a `Handshake` takes place or not, the next command is a Request. Every Request and Reply has a name, which is placed right after the keyword `Request` or `Reply`. Request represents a message from the

client intended for the server. It consists of sending a combination of integers, strings and bytes. Each type is sent separately in the order in which they are written in the *MSPL* program. The server code is also generated to accept the data structures in this order providing the necessary alignment. After all data has been sent, then Reply data structures are sent from the server to the client in the same way the Request message was sent from the client.

The language accepts as many Request–Reply statements as required by the protocol being implemented. For every Request there is zero or more possible Replies. An example of a request that may not need a reply is the `quit` command in the FTP protocol. It is also possible to name a Request with no parameters. This was done to easily handle more complex protocols in RFC. When no parameters are supplied, then bytes are sent. They are stored in a standard variable created in every message packet with the size of the field set to `buffersize`. After all the Request–Reply statements have been written, the keyword `End` is written which signifies the end of the *MSPL* program.

3.4 MSPL Parsing and Syntax Checking

The current *MSPL Compiler* is very basic, printing error messages that will help you find where an error may be and what might be the cause of it. If the *MSPL* program is not successfully compiled, then the code generation process never commences. Similar to the *Fabius* Compiler, the *Java* code is pre-generated with *holes* where values need to be inserted [Lee, 1996]. Details of the parser and *MSPL Compiler* are beyond the scope of this paper and they are described in [Douglas, 2000].

3.5 Generated Protocol Modules

Once the program written in *MSPL* passes through the Syntax Parser successfully, the Code Generation Process may begin. The process of code generation creates four main files as output. These files can be categorized as the message packet file, the client file, the server interface file and the server file.

3.5.1 Message Packet Architecture

Every variable declared in a Request statement or a Reply statement appears in the message packet structure. This message class is the return type of the generated functions. Figure 3 – 5 depicts the structure of an *ESFTP* message packet.

String Type	int Type	int Type	byte Type	String Type	String Type
Filename	statusRef	fileLength	actualFile	OverWrite	noFileError

Figure 3 – 5. ESFTP Message Packet Structure

The message packet sent is dynamic since not all the data types represented in Figure 3 - 5 are ever sent in one message packet. These are all the data types specified in the *MSPL* program written for *ESFTP* that will be required either for a request or a reply statement. Each data type is also assigned a variable name as shown below the dotted line in Figure 3 – 5. Depending on the Request made, the message structure will change dynamically to send only the necessary parameters for the specified request. The server code does the same for each reply sent back to the client. The client knows which reply to expect by checking a standard variable called the Reply name.

3.5.2 Client Protocol Module

The generated client module contains functions, which will be called by the user's client module to take care of low-level communication and the ordering that was embedded in *MSPL*. For example, in the *ESFTP* code shown earlier, a function called *put* would be generated with parameters `String` for the name of the file, `int` for the size of the file being sent and `byte[]` for the actual bytes of the file which are being sent to the server. All these parameters must be present when this function is called by the user's client code. The main advantage here over the common RPC, RMI and Corba code is that once this function is called, the work of receiving the reply to this request is also executed and a reply of `success` or an `error` is sent back to the user's client program in the form of a message, which contains several predefined fields that the user knows to check to get the relevant information needed. In other words, the client and server code is automatically aligned as described earlier in Section 2.9.

3.5.3 Generated Server Interface

The generated interface file is the interface between the generated server module and the user's server modules. The interface allows the user to not have to edit any of the generated code. The interface is extended using the `implements` command in *Java*. The advantage of using an interface file is that if for some reason, the code generated must be regenerated, then since the user did not modify the generated code, no extra coding or modifications by the user are lost.

3.5.4 Server Protocol Module

The next file generated is the server module, which calls the user's server program once it receives a message from the client side. This file receives messages from the client Request statements and sends data over the network connection for Reply statements.

Upon receiving data for a Request statement, it calls a function in the generated interface, which must be defined by the user's code. For example, if the `put` request is executed, then the generated server would call the `put` function in the interface class which must be implemented by the user. This is true because a server that implements a given interface promises to support all the methods defined by the interface. The client need not be concerned with how the server implements the interface. The Server Interface box in Figure 3 - 6 shows the interface class for the *ESFTP* example described throughout Section 3.

3.6 User-Written Modules

The user-written code is simplified greatly by writing a few lines in *MSPL*, which generates the communication code and also takes care of ordering. The main goal of the user's code is to manipulate the information it sends and receives from the client or server in order to carry out the task the application is supposed to do.

3.6.1 User-Written Client Modules

The user-written client modules import the generated client module, which then permits the user to call any functions in the generated client protocol code. The reply type of all the generated functions is

Message type. The user is responsible for checking the fields they asked to be created in the Message. For example, in *ESFTP*, if a Request Statement was `get` and it had the variable `filename` as a `String`, then in Message there would be a field of type `String` with the variable name `filename`. Now if the function returns type Message, which is stored in the variable `putReply`, then to access the `filename` field you would write `putReply.filename`.

3.6.2 User-Written Server Modules

The server-written code consists of functions that should be called depending on the Request Message received from the client. If the *ESFTP* `put` Request is sent to the server, then the generated server calls the `put` function of the user's server module with the message packet that was sent to it from the client side. This function is guaranteed to exist because of the generated interface that is implemented by the user's server module.

Client Protocol Module

```
public static messageType put(String filename,
                    int filelength, byt[] actualfile) {

  send.writeInt(toSocket.filename.length());
  send.writeChars(toSocket.filename);
  send.writeInt(toSocket.length);
  System.out.println("Sending bytes in actualfile");

  send.write(toSocket.actualfile);

  receive..read(fromSocket.replyName);

  return fromSocket;
}

public static messageType get(String filename) {

  send.writeInt(toSocket.filename.length());
  send.writeChars(toSocket.filename);

  receive..read(fromSocket.replyName);

  return fromSocket;
```

Server Protocol Module

```
stringLength = receive.readInt();

for (int i = 0; i < stringLength; i++) {
  fromSocket.filename += receive.readChar();
}

fromSocket.length = receive.readInt();

System.out.println("Receiving bytes");
receive.read(fromSocket.actualfile);

toSocket = GeneratedInterfaceInstance.put(fromSocket);

send.writeBytes(toSocket.replyName);
```

Server Interface

```
interface GeneratedInterface {
  public messageType get(messageType info);

  public messageType put(messageType info);
}
```

Generated Code

User-written Code

Client Module

```
info = ftp.put(fileName, length, theBuffer);


info = ftp.get(fileName);
```

Server Module

```
public messageType put (messageType info) {
  File theFile = new File(".", info.filename);
  System.out.println("USER >> Filename receiving:" +
                     "info.filename);
  try {
    FileOutputStream writeFile = new
    FileOutputStream(info.filename, true);
    System.out.println("USER >> writing " +
    info.length + " bytes!!!");
    writeFile.write(info.actualfile, 0, info.length);
    writeFile.close();
  }


  info.replyName = "successfull";
  return info;
}
```

Figure 3 – 6. Sample-Generated and User-Written Code for ESFTP

3.7 A Sample of Generated and User-Written Code for ESFTP

Figure 3 – 6 shows the code that is behind the boxes in Figure 3 – 2, representing Other Client Modules, the Client Protocol Module, the Server Protocol Module and Other Server Modules. The code shown for each module is a portion of the complete code that was written by the user or generated by the *MSPL Compiler*.

The `put` method is shown in Figure 3 – 6. First, the client may receive some data from the application user, requesting a file be copied from the local machine they are on, to another machine running the server application. The line `ftp.put(…)` calls the generated Client Protocol Module with the

specified parameters. Upon receipt of this call, the Client Protocol Module contacts the Server Protocol Module and sends the data across the network using the Message Packet described earlier in Section 3.5.1. Once the message packet arrives, the generated Server Protocol Module calls the `put` method through the generated Server Interface. It is then up to the user to extract the information from the message packet and place the appropriate data in a reply message packet. The `return` command gives control back to the generated Server Module, which then sends the reply message packet to the generated Client Protocol Module. The user-written Client Module originally called this module, so it returns a message packet type.

3.8 MSPL Library

The main thrust behind having a *MSPL Library* is to reduce the time of generating code. Calls to the *MSPL Library* are generated not the code. That is, repeated code does not have to be reproduced several times. Instead, these lines of code frequently used across many protocols, are abstracted into functions. For example, request and reply use the same basic communication concepts every time.

4. Implementation of RFC Protocols

As experiments for proof of concept and usability, basic parts of the *Simple Mail Transfer Protocol (SMTP)*, *Hypertext Transfer Protocol (HTTP)*, and the *File Transfer Protocol (FTP)* were implemented using *MSPL*. These protocols are widely used and are specified in *Request For Comments (RFC)*. Due to space limitations, only the implementation of HTTP RFC 2616 is described in this paper. The details of the SMTP and FTP implementations may be found in [Douglas, 2000]. After compiling the *MSPL* programs, sample user code was also written. Communication between the generated client code and an existing server which implements the same RFC was attempted as was communication between the generated server code with an existing client that implement the same RFC. The goal of testing a generated client with an existing server and a generated server with an existing client was to demonstrate that *"real world"* protocols can be specified in *MSPL* and the *MSPL Compiler* produces the appropriate code for communication.

4.1 Implementation of HTTP RFC 2616

HTTP is a request/reply protocol. A client sends a request to the server in the form of a request method, URI (Uniform Resource Identifiers) or URL (Uniform Resource Locator), and protocol version, followed by several lines with client information. The server replies with a status line, including the message's protocol version and a success or error code, followed by several lines with server information [Fielding et al., 1999]. HTTP communication usually takes place over TCP/IP connections. The default port is 80, but other ports can be used [Reynolds and Postel, 1994].

For the purpose of a functional client and or server, the only methods required were the `GET` and `QUIT` methods. Figure 4 – 1 shows the *MSPL* code used to generate the java protocol modules.

```
1.    Parameters
2.       defaultClientPort 55000,  # between 0 and 65535
3.       defaultServerPort 55000,  # between 0 and 65535
4.       bufferSize 1024,          # same size buffer for Client and Server
5.       maxClientsSupported 10;

6.    Begin
7.       Request Get
8.          Constant String "GET ",
9.          String filename,
10.         Constant String " HTTP/1.1 \r\nAccept: ",
11.         String accept,
12.         Constant String "\r\nAccept-Language: ",
13.         String acceptLanguage,
14.         Constant String "\r\nAccept-Encoding: ",
15.         String acceptEncoding,
16.         Constant String "\r\nUser-Agent: ",
17.         String userAgent;
18.         Constant String "\r\nHost: ",
19.         String hostname,
20.         Constant String "\r\nConnection: ",
21.         String connection,
22.         Constant String "\r\n\r\n";       # Two CRLF's to end request
23.      Reply Ok # successfully received, understood, and accepted
24.         Constant String "HTTP/1.1 200 OK \r\n",
25.         Constant String "Server: ",
26.         String serverType,
27.         Constant String "\r\nDate: ",
28.         String currentDate,
29.         Constant String "\r\nContent-type: ",
30.         String contentType,               # i.e. text/plain
31.         Constant String "\r\nLink: ",
32.         String link,
33.         Constant String "\r\nEtag: ",
34.         String etag,
35.         Constant String "\r\nLast-modified: ",
36.         String lastModified,
37.         Constant String "\r\nContent-length: ",
```

```
38.          String contentLength,
39.          Constant String “\r\nAccept-ranges: “,
40.          String acceptRanges,
41.          Constant String “\r\n”,
42.          byte[] actualFile,
43.          Constant String “\r\n\r\n”;
44.       Reply fileNotFound        # The request contained bad
45.          Constant String “400 file not found \r\n\r\n”;
46.       Reply serverNotAvailable  # The server failed to fulfill
47.          Constant String “500 server not available \r\n\r\n”;  # an invalid request
48.    End
```

Figure 4 – 1. MSPL Code for HTTP

The generated client was also linked to user-written modules to produce the client software. This is the only example where `maxClientsSupported` was tested extensively and worked well. Most web browsers developed now automatically request several connections to the same server in order to speed up the time required to download a web page that has several pictures. According to line 5 of the *MSPL* code, up to 10 connections can be made to the server at once. Line 8 show the first line that is sent for the request `Get`. It is followed by a filename which is declared as variable on line 9. Lines 10 through 15 send the server information about formats of data that the client understands. Lines 16 through 20 complete the `Get` request by sending the server information about which host the client is located on and the current state of the connection between the client and server. As shown on line 22, the end of a request is signified by a double *CRLF* (i.e. \r\n\r\n). For the `Get` request there are three possible replies which are `ok, fileNotFound,` or `serverNotAvailable.` Lines 23 through 41 show the information about the server and the file requested that is sent back to the client when the server establishes the request is *ok*. Line 42 is where the actual file is sent to the client and the following line signals the end of the reply. The latter two replies are used when there is an error of some sort. Each sends one line of text to the client that is prefixed with a number. The number is used to represent the general type of error that has occurred. The purpose of these numbers are further described in Figure 4 – 3.

4.1.1 HTTP Server Software

For the implementation of the generated code of the HTTP RFC 2616 protocol, the generated server was tested with the Microsoft Internet Explorer Version 5.00.2314.1003 client. The server was set up to run

on port 55000 instead of port 80 where *HTTP* servers usually run. To direct the *HTTP* client to my server instead, the address and port had to be written in the address window as shown below:

http://winnie.fit.edu:55000/~mdouglas

The generated server was able to send both graphics and text back to the client, which was then able to display them. In this example, the `buffersize` entered in *MSPL* played a major role. Depending on the `buffersize`, the time to load a standard 8½ by 11 inch page with one or two pictures varied by over 5 seconds.

There are several other commands that were not implemented but the `GET` request was sufficient to successfully transfer files and images between the client and server being used. A script of the conversation is shown in Figure 4 – 2. The lines in bold are the relevant pieces of information that are currently being used to service the requests.

```
1.      Script started on Wed Mar 29 01:56:24 2000
2.      /usr/users/student/mdouglas/public_html> java httpd
3.      Accepting connections on port 55000
        ……
5.      Connection Established!!!
6.      Request: get
7.      USER HTTPD >> *** BEGIN HTTP Packet:
8.      GET /~mdouglas HTTP/1.1
9.      Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-
        powerpoint, application/vnd.ms-excel, application/msword, application/pdf, */*
10.     Accept-Language: en-us
11.     Accept-Encoding: gzip, deflate
12.     User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows NT; DigExt)
13.     Host: fit.edu:55000
14.     Connection: Keep-Alive
15.     USER HTTPD >> *** END HTTP Packet.
16.     USER HTTPD >> pathname: '/~mdouglas'
17.     USER >> Filename sending:index.html
        ……
20.     USER HTTPD >> Finished Sending File!!!
        ……
25.     Request: get
26.     USER HTTPD >> *** BEGIN HTTP Packet:
27.     GET /Images/Flagbda.gif HTTP/1.1
        ……
29.     Referer: http://fit.edu:55000/~mdouglas
        ……
33.     Host: fit.edu:55000
34.     Connection: Keep-Alive
35.     USER HTTPD >> *** END HTTP Packet.
36.     USER HTTPD >> pathname: '/Images/Flagbda.gif'
37.     USER >> Filename sending:Images/Flagbda.gif
        ……
40.     USER HTTPD >> Finished Sending File!!!
        ……
54.     Connection: Keep-Alive
        ……
65.     ^C
        /usr/users/student/mdouglas/public_html> exit
66.     script done on Wed Mar 29 01:58:45 2000
```

Figure 4 – 2. HTTP Server Conversation Script

Line 2 in Figure 4 – 2 shows how the program is executed. The *daemon* is started by running the generated Server Protocol module. On line 5, a connection is accepted from a client, which is the Internet Explorer client. The first request received is in bold on line 8. Every request consists of several lines. A line without text on it, known as *carriage-return line-feed (CRLF)*, denotes the end of a request. RFC 2616 for HTTP also suggests that in the interest of robustness, servers should ignore any empty lines received where a Request-Line is expected [Fielding et al., 1999].

The Request-Line begins with a method token, followed by the Request-URI, the protocol version, and ends with a *CRLF*. The tokens are separated by <*SP*> (space) characters. Except in the final *CRLF*, no *CRs* or *LFs* are allowed. The following line shows the protocol for a Request-Line:

Request-Line = Method<SP> Request-URI <SP> HTTP-Version<CRLF>

On line 8 in the HTTP conversation script, *GET* is the Method, /~mdouglas is the Request-URI and HTTP/1.1 is the HTTP-version. The entire HTTP packet sent for the first request spans from line 8 to line 14. The Method token indicates the method to be performed on the resource identified by the Request-URI. It is also worth noting that the method is case-sensitive.

Line 13 is the next portion of data that was used to service the `GET` request. A client must include a Host header field in all HTTP/1.1 request messages. If this line is not in the request message then all standard HTTP/1.1 must respond with a 400 (Bad Request) status code [Fielding et al., 1999].

A second web page is requested on line 26. This file is referred to by a link on the current page, therefore, line 29 is sent to inform the server of this fact. The Referrer request-header allows a server to generate lists of back-links to resources that can be used for logging or optimized caching. The protocol for a Referrer is: *Referrer = "Referrer" ":" (absoluteURI | relativeURI)*

In HTTP/1.0 [Fielding et al., 1999], most implementations used a new connection for every request/response exchange. In HTTP/1.1, a connection may be used for one or more request/response exchanges. Lines 14, 34 and 54 inform the server of whether the connection should be closed or not. A Connection, however, may be closed for a variety of other reasons.

After receiving and interpreting a request message, the server responds with an HTTP response message for which the protocol is: *Response = Status-Line <CRLF> [message-body]*

The Status-Line is the first line in the Response message and it consists of the protocol version followed by a numeric status code and an optional text message describing the status code. The protocol for the status-line is: *HTTP-Version <SP> Status-Code <SP> Reason-Phrase <CRLF>*

Figure 4 – 3 shows and gives a brief description of the five main Status-Code categories. The first digit of the Status-Code defines the class of response. The last two do not have any categorization role but may be used by the programmer for more specific meaning.

- `1xx: Informational - Request was received, and now continuing process.`
- `2xx: Success - The action was successfully received, understood, and accepted.`
- `3xx: Redirection - Further action must be taken in order to complete the request.`
- `4xx: Client Error - The request contains bad syntax or cannot be fulfilled.`
- `5xx: Server Error - The server failed to fulfill an apparently valid request.`

Figure 4 – 3. HTTP Status Codes [Fielding et al., 1999]

After the status-line, there is a *CRLF* and then the message body, which in the case of the `GET` request, is the actual bytes of the file that was requested.

4.1.2 HTTP Client Software

For HTTP RFC 2616, the client was also generated. In Figure 4 – 4, the client is shown interacting with a *Netscape-Enterprise/3.5.1G* server. For the purpose of this example, a one-line web page is requested and sent back to the client. The client prints the one line directly to the screen. This is the line that would usually be displayed by a graphics enabled web browser such as Netscape Explorer or Internet Explorer.

```
1.   Script started on Sat Apr 15 23:59:31 2000
2.   CS:1>> java userHTTP
3.   Starting HTTP Application...
4.   Connect Address: fit.edu
5.   Enter Address: /~mdouglas/oneLine.html
6.   Packet being sent:
7.   GET /~mdouglas/oneLine.html HTTP/1.1
8.   Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-
              powerpoint, application/vnd.ms-excel, application/msword, application/pdf, */*
9.   Accept-Language: en-us
10.  Accept-Encoding: gzip, deflate
11.  User-Agent: Mozilla/4.0(compatible; MSIE 5.0; Windows NT; DigExt)
12.  Host: maelstrom.cs.fit.edu
13.  Connection: Keep-Alive
```

```
14.  Executing get function
15.  Finished sending Request Parameters
16.  Returning control to user...
17.  Reply packet for Request:
18.  HTTP/1.1 200 OK
19.  Server: Netscape-Enterprise/3.5.1G
20.  Date: Sun, 16 Apr 2000 04:03:31 GMT
21.  Content-type: text/html
22.  Link: <http://winnie.fit.edu/~mdouglas/oneLine.html? PageServices>; rel="PageServices"
23.  Etag: "240fef-b-38f8c9d2"
24.  Last-modified: Sat, 15 Apr 2000 19:58:10 GMT
25.  Content-length: 40
26.  Accept-ranges: bytes
27.  Hi there, this is a one line web page
28.  Enter Address: quit
29.  Finished sending Request Parameters
30.  Returning control to user...
31.  Thank you for using M.E. (Melvin's Explorer)
32.  CS:2>> exit
33.  script done on Sat Apr 15 23:59:58 2000
```

Figure 4 – 4. HTTP Client Conversation Script

Figure 4 – 4 shows a successful conversation between the generated client and a commercial *MS Internet Explorer* server. On line 2, the client is invoked. Lines 4 and 5 allow the user to specify a specific web page they would like to *browse*. Lines 6 to 13, show the entire request packet sent. It informs the server of what the client is capable of supporting such as what picture formats are recognized. Line 7 is the first line read by the server and identifies the service being requested. In this case, it is the `GET` method. The syntax for this command is: GET <SP> URI.

This line along with line 12, informs the server of where to locate the file being requested. The information of what host the client is running on is obtained at run-time using the standard hostname command, which returns the information shown in italics on line 12. In HTTP versions 1.1 and higher, line 13 is used to inform the server on whether to close the connection or keep it open. The last request should say "*Connection: close"* informing the server to complete the request and close the connection. Lines 14 to 17 are printed by the generated client module to inform the user of what is happening. Line 18 is sent from the server to the client confirming the request was received and processed OK. The number 200 implies the request was semantically correct and successfully serviced. This line along with the following lines to line 26, are known as the *header*. They inform the user about the file, which is about to be sent. The date of request, the type and size of file, and the last date of modification of the file are among the more important pieces of information supplied to the client. In a more complex client, this

information may be used to decide whether the file needed to be transferred to the client at all or if the client could simply retrieve the file from its cache. Line 27 is the actual data in the file *oneLine.html*. The following lines are used to gracefully close the connection to the server and exit from the client application.

5. Concluding Remarks

We have made several initial definitive steps in the right direction, we believe, to increase the level of quality in the development of client-server software. *MSPL* proved to be useful not only in non-standard protocols like *ESFTP*, but also in standard protocols like SMTP RFC 821, HTTP RFC 2616, and even FTP RFC 959. This research could have a significant impact on the development of future network code generation applications and protocol specification languages. Most network applications in the past have concentrated on providing function calls. This research, however, looks more closely at how generated code can make use of ordering that is embedded in protocols. There has already been a substantial impact in the area of re-use of code by other research and this research shall at least add more arguments for re-use of code.

A strength of *MSPL* is it is easy to read and understand. The way in which the syntax parser was implemented makes it fairly easy to extend the language to entail new features. This was the case when the `Handshake` command was added to the language. *MSPL* seems pretty easy to use although there have not yet been many users of the system. The independent development of *MSPL* from the compiler makes this solution quite portable since a compiler from *MSPL* to any programming language can easily be developed.

There are several prospects for future work, some of which are currently being worked on. The primary future work that needs to be done in order to support RFC 959 File Transfer Protocol, is to allow more than one connection between a client and a server. This leads to more problems that must be thought through and tackled. For example, in FTP RFC 959, the port for the *data connection* can change several times in one session as it is only open long enough to service one Request. Once it closes and

reopens again for another Request, it is quite possible and likely that a different port will be used. This implies the port would have to be changed more than once during the execution of the application. This leads to the next question of whether it is worth changing the language to allow the user to change the port from the user's code. This method could be placed in the *MSPL Library*. Another direction would be to provide more error handling features similar to the BEA Tuxedo package described in Section 2. This would greatly increase the reliability of the language when used in the *real world*. Other directions include allowing optional fields and order-insensitive fields to be specified in *MSPL*, nested request-reply structures for conversations spanning across multiple request-reply interactions, and extending our implementation to specify the remaining parts of the three standard protocols studied here and other standard protocols.

References

[Amer et al., 1997] P. D. Amer, A. S. Sethi, M. Fecko, and M. Uyar, "Formal Design and Testing of Army Communication Protocols Based on Estelle," *Proc. 1st ARL/ATIRP Conf.*, pp. 107-114, 1997.

[BEA, 1996] BEA, "Programming a Distributed Application: The BEA Tuxedo® Approach", White Paper 1996. http://www.bea.com/products/tuxedo/paper_distributedapp.html

[Douglas, 2000] Melvin A.L. Douglas, "*MSPL: A Protocol Language For Generating Client-Server Software*," MS Thesis, Dept. of Computer Sciences, Florida Institute of Technology, May 2000.

[Engelen et al., 1996] Robert A. Van Engelen, Lex Wolters, and Gerard Cats, "Automatic Code Generation for High Performance Computing in Environmental Modeling," *Proceedings of the 1996 EUROSIM International Conference on HPCN Challenges in Telecomp and Telecom: Parallel Simulation of Complex Systems and Large-Scale Applications*, June 10-12, 1996.
http://www.wi.leidenuniv.nl/home/robert/

[Fielding et al., 1999]R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, "Hypertext Transfer Protocol Request For Comments (RFC) 2616," June 1999.

[Kohler et al., 1999] E. Kohler, M. F. Kasshoek, and D. R. Montgomery, "A Readable TCP in the Prolac Protocol Language," *Proc. SIGCOMM99*,1999.

[Reynolds and Postel, 1994]J. Reynolds and J. Postel, "Assigned Numbers", STD 2, RFC 1700, 1994.

[TanenBaum, 1996]Andrew S. TanenBaum, C*omputer Networks*, Third Edition, pp. 28-29, 1996.