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

ABSTRACT

To ease the burden of repeatedly writing low-level communication and protocol code for client-server software, we seek to design a protocol language, "*My Simple Protocol Language*" *(MSPL)*, that produces the corresponding communication code. The programmer then supplies the rest of the application-specific code. The main contribution is that, unlike RPC, Corba or RMI, we provide the user with not only functions that are for communication, but also for protocols.

KEY WORDS: Software Development, Protocol Language, Code Generation.

1. INTRODUCTION

Client-server software has become increasingly critical due to the popularity of the Internet. Implementing the underlining protocols requires programming low-level protocol and communication details, which is time-consuming and reduces maintainability. Furthermore, changes in the protocol specification demand matching revisions on both the client and server programs.

In this paper we focus on two problems. The first is providing a protocol language that allows writing client-server software efficiently and reliably. The second is to demonstrate the feasibility of using the protocol language on '*real world*' protocols like HTTP RFC 2616.

Section 2 gives an extended overview of related work. Section 3 details our proposed language *MSPL* and its architecture. Section 4 analyzes the use of *MSPL* to develop clients and servers that can interact with real-world servers and clients. Section 5 summarizes our contributions and potential improvements.

2. RELATED WORK

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 [1], which specifies protocols as a set of finite state machines. Though Estelle eases the formal verification process by machines, it is not easy for humans to read. In addition, it was not originally designed to facilitate the automated process of implementing the specified protocols. Languages like Prolac [2], 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 or lower layers in the OSI model because they constitute the vital infrastructure of the Internet. However, languages for specifying protocols above the Transport layer are not well investigated, even though many distributed and network applications, particularly in the client-server model, have been developed. 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.

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 [3]. Tuxedo and *MSPL* both basically aim at developing client-server software at 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. While Tuxedo supports multiple types of *send* and *receive* commands, *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.

RPC is based on the request/response paradigm, modeled after the local procedure call structure. RPC intertwines procedures with protocols and demands procedural thinking. However, *MSPL* advocates a declarative approach in specifying the protocols instead of

implementing them. 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, the same protocol needs to be ported to the new *target language*.

2.1 Library/Specification-Based Approaches

Developing client-server software at a higher abstraction level 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 abstracted routines. *MSPL* provides a higher level of 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 hide low-level details from the programmer. Another advantage is the fact they shorten the development time. 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.

Another major advantage of the specification-based approach is that protocols are automatically "aligned". By alignment, we mean that if the number, format, or order of parameters for a particular request is changed on the client side, then the server side is automatically adjusted to match 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. 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 designing *'My Simple Protocol Language' (MSPL)*, which is used to write programs that implement the communication protocols. Figure 3-1 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. 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. The *MSPL* Compiler is currently implemented in Java and the target language for code generation is Java as well.

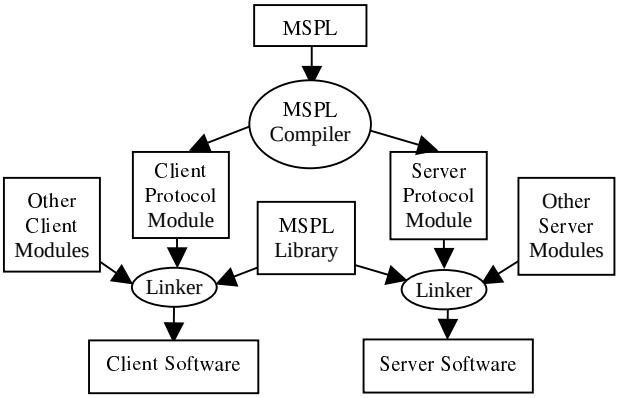

Figure 3–1. MSPL Architecture

3.1 ESFTP

Before we describe exactly how to write a program in *MSPL* and how the client and server code is generated, we introduce a simple protocol called the *Extremely Simple File Transfer Protocol (ESFTP)*, which will be used as a running example throughout this section. It is a much simpler version of the RFC 959 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; details are in [4]. The protocol is also used to send error messages between the client and server.

3.2 Implementing ESFTP in MSPL

In this section, we examine how to write a program in *MSPL* and how the client and server code is generated. In Figure 3-2 *ESFTP* is specified in *MSPL*

All the parameters have default values, which can be overridden. 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 on Definable Communication Parameters.

Comments may be inserted by preceding the text with a number # sign.

```
1.    # MSPL file used to generate code for the ESFTP
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 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 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 3-2. Specification of ESFTP in MSPL

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`. 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 18 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 through 23. The two possible replies to this request are `Successful` or `fileExists`. `Successful` is the name of the reply on line 24 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 26 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 and does not expect a reply. The `quit` command is not written in Figure 3-2 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 [4].

3.3 Definable Communication Parameters

In Figure 3-2 the section between the keywords `Parameters` and `Begin`, Lines 2 to 7, 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.

Also in order to handle unpredictable disconnections of mobile computing platforms there is a variable called `timeout`. This allows one to specify the amount of time before a standard message is displayed by the client when it is unable to send or receive from a server due to network delay or disconnection. The message informs the user the connection cannot be established and gives them the option to retry right away or later. If they decide to retry then the entire request will be resent to the server.

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. 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 [3].

3.4 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 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.

3.5 Compiler and Generated Modules

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 [5]. Details of the parser and *MSPL Compiler* are beyond the scope of this paper and they are described in [4].

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. The structure of an *ESFTP* message packet is dynamic since not all the data types represented are ever sent in one message packet. The possible data types are Strings, integers and bytes. 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 allowing us to send more than one value of the same data type. 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 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. That is, the client and server code is automatically aligned as described earlier in Section 2.1.

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.

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.

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`.

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.

3.7 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 communication concepts every time.

Due to space limitation, we have not included a sample of the generated code and the conversation script recorded while the client and server were running. This can be found, however, in [4].

4. RFC PROTOCOL IMPLEMENTATIONS

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 [4]. 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 [5]. HTTP communication usually takes place over TCP/IP connections. The default port is 80, but other ports can be used [6]. 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,          # 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";      #CRLF's end request
23.     Reply Ok # successfully received 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  # failed to connect
47.      Constant String "500 server not available \r\n\r\n";
48.     End
```

Figure 4-1. Part of HTTP specified in MSPL

The generated client was 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. Line 5 specifies that up to 10 connections can be made to the server at once. The `Get` request starts on line 7. Lines 10 through 17 send the server information about data formats that the client understands. The `Get` request is completed in lines 18 through 22. The end of a request is signified by a double *CRLF* (i.e. \r\n\r\n) as shown on line 22. There are three possible replies to the `Get` request which are `ok, fileNotFound,` or `serverNotAvailable.` Lines 23 through 41 shows the server's preferred data formats and the file format which is sent back to the client when the server determines 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 a string to the client that is prefixed with a number to represent the general type of error that has occurred.

From the generated code of the HTTP protocol specification, the generated server was tested with the Microsoft Internet Explorer Version 5.0 client. The server was set up to run on port 55000 instead of the default http port which is 80. The generated server was able to send both graphics and text back to the client which displayed them. Depending on the buffer size entered in the *MSPL* code, the time to load a standard 8½ by 11 inch page with a picture varied by over 5 seconds. There are several other commands that were not implemented but the GET request was sufficient to transfer files and images between the client and server. In addition, the generated client was successfully tested with a real server. Due to space limitation, we have not included a sample of the generated code and the conversation script recorded while the client and server were running. This can be found, however, in [4].

5. CONCLUSION

We have made several initial definitive steps in the right direction, we believe, to increase the 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 FTP RFC 959. Many existing tools concentrate on providing function calls. This research, however, focuses on how to generate code from a protocol specification. Part of *MSPL's* strength is its readability. *MSPL* allows and promotes more efficient development of reliable client-server software. *MSPL* seems relatively 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 target programming language can be developed.

There are several prospects for future work. One direction is to extend *MSPL* to support more than one connection between a client and a server, which are allowed in RFC 959 File Transfer Protocol. This leads to additional issues that have to be addressed. 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 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 it is 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

[1] 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.,* 1997, 107-114.

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

[3] BEA, Programming a Distributed Application: The BEA Tuxedo® Approach, White Paper,1996.

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

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

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