MSPL: A Protocol Language For Generating Client-Server Software

by
Melvin Austin Leroy Douglas

Bachelor of Science
in Computer Science
Florida Institute of Technology
1998

A thesis
submitted to the Graduate School of
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
May, 2000

We the undersigned committee hereby recommend that the attached document be accepted as fulfilling in part the requirements for the degree of Master of Science in Computer Science.

"MSPL: A Protocol Language For Generating Client-Server Software"
a thesis by Melvin Austin Leroy Douglas

Philip K. Chan, Ph. D.
Assistant Professor, Computer Science
Thesis Advisor

Ryan Stansifer, Ph. D.
Associate Professor, Computer Science
Committee Member

Palmer C. Stiles, M.S., P.E.
Assistant Professor, Mechanical Engineering
Committee Member

William D. Shoaff, Ph. D.
Associate Professor and Program Chair
Computer Science

Acknowledgements

First and foremost, I would like to thank God, My Lord and Savior, for giving me strength, hope and perseverance in my studies. Without Him none of this would be possible or worthwhile.

I would like to thank my major advisor Dr. Chan for his innovative ideas and constructive criticism that helped make this research a success. Thank you also for your prodding towards excellence with the freedom to choose and conduct my thesis towards my interests. To my Committee Members, thank you for taking the time to read and revise my written and oral presentation of this thesis.

I would also like to thank my Mother, Father and Sister for their prayers and support throughout my educational endeavors. I am very grateful to have a family who has encouraged me each and every step of my life. To a very special friend, Adolé Tounou, thank you for your motivation and help to remain focused on the life-size picture. I am indebted to many friends and colleagues whose love and support have been a constant source of inspiration for me.

Dedication

To My Heavenly Father who has showed me so much Love and Kindness.

Abstract

MSPL: A Protocol Language For Generating Client-Server Software

by

Melvin Austin Leroy Douglas

Thesis Advisor: Philip K. Chan, Ph. D.

Client-server programs are 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. It is worth noting the programmer 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 which are specified in *MSPL* programs. The *MSPL* programs are then passed to the 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.

Table of Contents

List of Figures

Chapter 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. The British Aerospace Dependable ComputingSystems Center is looking at how formal techniques can be employed to ensure that an automatic code generator produces code that is faithful to its specification. The use of formal techniques is important to this process since it is only through these thatthe high level of assurance required can be attained. The goal is to attain this assurance while placing as few requirements on the programmer as possible. 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.

1.1 Problem Statement

There are two problems that I 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. The system is built in *Java*, which sacrifices efficiency for portability to some extent

1.2 Organization of Thesis

The introductory chapter describes the statement of the problem as well as a proposed solution. Chapter 2 gives an extended overview of related work. Techniques used in this area of research in the past are discussed and compared. Chapter 3 looks more closely at the solutions to the problems being focused on in this thesis. It explains different concepts used during the development of the thesis. An example of how *MSPL* may be used for the implementation of a user's protocol is discussed and analyzed. Chapter 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. Chapter 5 discusses the conclusions made after in-depth research, implementation and testing of the generated code.

Chapter 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.

2.1 A Brief History

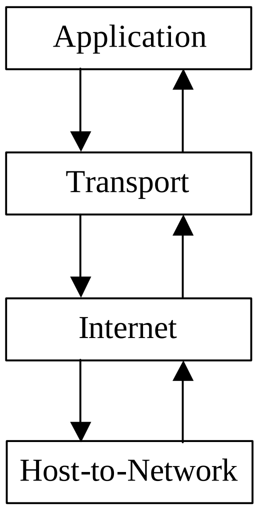
In this section, several existing systems that use code generation are compared. A comparison and contrast of their protocol compositions are also given.

Traditionally, protocol compositions have mainly been static in that compositions are determined at compilation time such as the *TCP/IP* stack, which is one of the more popular static compositions.

While the *TCP/IP* stack works well for simple cases, it has weaknesses when it encounters demanding clients or rich networking environments. This is mainly because characteristics of the network are not known until runtime. There are two main shortcomings to this static protocol approach. The first is the exponential code growth inherent in it. For example, to perform data conversion between two hosts, a static system must pre-compose all possible conversion methods. The second subtler problem of static composition is that it is a closed system.

At the other extreme, runtime or dynamic composition can be used to combine only those protocol stacks that are needed. The flexibility of dynamic protocols, however, prevents us from being able to integrate different layers of the system. The benefits of this method are that protocols can be written in any language and the protocols can be compiled separately. Protocols can even be dynamically linked as their implementations are upgraded. Oriented vertically, the low end of the stack is the link layer protocol and the high end is the application layer protocol as shown in Figure 2 - 1.

Figure 2 - 1. TCP/IP Reference Model



The central efficiency problem with modular protocol design is that separation of protocol levels prevents integration of each protocol's data manipulations [Clark, 1990]. Consider a two-layer stack consisting of TCP (Transmission Control Protocol) and RPC (Remote Procedure Call) protocol that guarantees correct byte order. When a message is received and delivered to TCP, the TCP layer will *touch* the entire message. Multiple message traversals are expensive given the current difference between memory and CPU (Central Processing Unit) speeds. One of the goals of a protocol is to provide support to allow each part of a message to be *touched* only once.

Dynamic code generation is the generation of executable code at runtime. This has become a popular topic but it is still used only by a minority of implementers [Hsieh, 1996]. Like static compilation, dynamic compilation can be used to eliminate interpretation. Run-time code generation has led to notable performance improvements in the areas of operating systems, simulators, graphics, matrix multiplication and dynamically typed languages. Current compilers do not optimize networking expressions well. This is mainly because there isn't a clear way in any language of writing these common networking operations such as checksumming. Protocol characteristics like complicated flow control, make protocol modules hard to read, verify and maintain. Specialized languages are a promising solution to this problem and compilers have been an active research area for decades [Kohler, 1999].

2.2 Protocol Description Techniques

Most existing protocol languages focus on verification. *Prolac* is a new statically typed object-oriented language that has been tailored for network protocol implementation and generates completely order independent *C* code [Kohler, 1999]. It resembles object-oriented languages like *C++* and *Java* but it is designed to be more useful than these languages for network protocols. *Prolac* is an expression language like *Lisp* and *ML*. Instead of verification, *Prolac* was designed for readability, extensibility, and '*real world*' implementation. The implementation is as modular as protocol processing is logically divided into minimally interacting pieces. As ideas were gathered from other specific languages designed with protocols in mind such as parallelism to model both sides of a connection, it often worked against readability, implementability, extensibility or all three. *Prolac's* final design is less domain specific than these languages.

Two protocol languages, or description techniques originally designed for developing OSI protocol suite are *LOTOS* and *Estelle* [Kohler, 1999]. *Estelle* structures a protocol as a set of finite state machines running in parallel and communicating with broadcast signals. This makes it very difficult to read. It is, however, a great method for test generation or state analysis. Even with carefully layered protocols, *Estelle* specifications would be very difficult to modify. Later, this protocol was improved dramatically just by removing its asynchronous parallelism which made it a completely sequential language. *RTAG* is a model that uses context free attribute grammars. It is considered to be easier to read than *LOTOS* and *Estelle*.

The *Prolac* compiler compiles *Prolac* into *C*. The high level *C* introduces relatively few temporary variables. Compilation time of complex implementations such as *TCP* take less than a second on a 266MHz Pentium II laptop [Kohler, 1999]. Inlining, path inlining and outlining all improve the efficiency of a protocol and are all used in *Prolac*. Inlining is replacing a function call with the function's body. Path inlining is recursive inlining while outlining is moving code for uncommon cases out of common case code. The *Prolac TCP* implementation consists of one-third the number of lines the Linux 2.0 TCP implementation has but theirs has more functionality. Figure 2- 2 shows a comparison of processing time and latency for an echo test. The test machine sends 4 bytes of data to an unmodified Linux 2.2.7 machine's echo port and waits for an acknowledgement. Results were averaged over five trials, each consisting of 1,000 round-trips, for a total of 10,000 packets (i.e. 5,000 input and 5,000 output). Processing time represents the average number of cycles it took to process a packet. The test machines were 200 MHz Pentium Pro desktops and they communicated over an otherwise idle 100Mbps Ethernet with one hub.

Figure 2 - 2. Micro-Benchmark Results for an Echo Test [Kohler, 1999].

	End-to-end latency (μs)	Processing time (cycles)
Linux TCP	184	3360
Prolac TCP	181	3067
Prolac without inlining	228	6833

Currently, one of the main weaknesses of *Prolac* is it is not as reliable as *Linux TCP* but this can change in the near future as the goal is to use it in '*real world*' situations. Besides having a good protocol implementation, the actual code generator needs to be made more efficient and portable.

2.3 Optimizing Communication by Aggregation

The research goals of one group from Stanford University were to optimize communication by eliminating redundant communication and aggregating small messages into larger messages [Amarasinghe, 1993]. They overlapped communication latency with computation where possible. To minimize communication cost, the Stanford SUIF compiler tries to maximize the intervals between communication. All the data needed within the interval are sent in one message. Their technique is based on an exact data-flow analysis on individual array element accesses. Unlike data dependence analysis, this analysis determines if two dynamic instances refer to the same value, and not just the same location.

Using this information, their compiler can handle more flexible data decompositions and find more opportunities for communication optimization than systems based on data dependence analysis. The *Last Write Tree* (LWT) information allows them to eliminate redundant data transfers. The LWT analysis automatically partitions the read instances into sets that share similar communication characteristics. This partitioning makes generating code routine and it also enhances optimizations [Amarasinghe, 1993]. Their model and techniques are useful in both value-centric and location-centric approaches. The scope of their technique is limited to programs consisting of a set of loop nests or conditional statements.

For example, suppose we need to merge the following loops [Amarasinghe, 1993]:

```
For I = 0 to 200 do
   Receive(…)
For I = 100 to 300 do
   Send(…)
```

Instead of generating one for loop with two conditional *if* statements as shown below:

```
For I= 0 to 300 do
   If 0 <= I and I <= 200 then
      Receive(…)
   If 100<=I and I<=300 then
      Send(…)
```

They can generate three consecutive for loops without any conditional *if* statement as shown below:

```
For I=0 to 99 do
   Receive(…)
For I=100 to 200 do
   Receive(…)
   Send(…)
For I=201 to 300 do
```

```
                    Send(…)
```

To generate the complete code for a processor, it is necessary to merge a processor's computation code, and its *receive* and *send* code for each communication set. In the original code, it is beneficial to merge the *For* loops because there is overlap in the work done for the value of *I* between 100 and 200 inclusively. If only one *For* loop is generated, then there must be two conditional *if* statements in the *For* loop to check the value of *I* each iteration. In the alternatively generated code, there are three sequentially placed *For* loops. By splitting the original code in this way, there is no need to add any conditional *if* statements or have overlap of *I* iterations between 100 and 200 inclusive.

The algorithm they developed that allows them to merge multiple nested loops together is called *loop splitting*. If the relative magnitude between the bounds of the individual loops is not known at compile time, *loop splitting* can expand the program size by a significant amount. Therefore, the SUIF compiler only uses *loop splitting* on inner loops or when the relative magnitudes between the loop bounds are known [Amarasinghe, 1993].

2.4 Fabius Compiler

At Carnegie Mellon University, the *Fabius* compiler was developed. *Fabius* takes ordinary programs written in a subset of *ML* and automatically compiles them into native code that generates native code at run-time. The dynamically generated code is often much more efficient than the statically generated code because it is optimized using runtime values. Although not every program benefits from run-time code generation, there has been little trouble finding realistic programs that run significantly faster, sometimes by more than a factor of four [Lee,1996]. The main focus of the *Fabius* system was on low-level optimization and code generation issues.

2.5 CTADEL System

The *CTADEL* system 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 [Engelen, 1996]. Efficiency and portability are generally conflicting goals. In general this results in several platform-specific versions of code. This is not advantageous from a maintenance point of view. Adding improvements to a model that is platform-specific is very difficult because improvements for one platform may be a step backward on other platforms.

Libraries are great tools to help increase portability but the arrival of new hardware platforms requires the redesign or at least extensive recoding of libraries in general. By taking advantage of specific hardware characteristics of the target computer architecture, portability and code-consistency problems are made absent. For each machine, an efficient hardware specific version of the code can be generated. *CTADEL* was developed with the goal of generating efficient code [Engelen, 1996]. In contrast to other systems, *CTADEL* takes the characteristics of the target computer architecture into account, providing the necessary information for the system to generate high-performance code for various computer architectures from a high-level language description model. The Latex package of the *CTADEL* system automatically generates reports of the code generation process, which is an advantage and strength it has over other systems. Optimization techniques used by *CTADEL* include algebraic simplification and global common sub-expression elimination. A trade off between the reduced computational complexity and the additional memory usage plays an important role in the generation of efficient code by the *CTADEL* system. From a software engineering point of view, a code generator can assist the programmers and relieve them from the task of coding efficient implementations for several hardware architectures.

Dynamic code generation allows aggressive optimization through the use of run-time information. At Massachusetts Institute of Technology, they developed a *Dynamic Code Generation* system *(DCG)* that does one pass code generation, is easily re-targeted and extremely efficient. One of the main weaknesses of code generation is that it costs approximately 350 instructions per generated instruction. This is the highest number of instructions per generated instruction out of all the systems researched. Dynamic code generation does not change the existing code, but rather augments it, enabling programs to create specialized instruction sequences based on runtime information. *DCG* efficiently generates executable code at runtime [Hsieh, 1996]. They focus on a demonstration of efficient, dynamic machine code generation from a machine independent specification. In 1994, this was the only stand-alone and easily retargeted dynamic code generator to emit binary instructions directly. To make client programs portable, they specify code using a machine-independent intermediate representation (IR) that is passed to *DCG*. To help maintain simplicity, they used the already tested interface of the *lcc* compiler [Fraser, 1991]. *DCG's* code generator is able to link directly to *lcc's* front-end. Testing its correctness consists of simply compiling existing test-suites to *Assembly* language, and testing the resultant output [Hsieh, 1996]. The interface is fully documented in [Fraser, 1991]. One form of optimization used is *Strength Reduction* where multiplication is replaced with shifts and adds.

2.6 Advantages and Disadvantages of Dynamic Code Generation

In some systems, code generation at run-time was very high, to the point where improvements gained by delaying compilation to run-time were eliminated by the cost of run-time compilation. For example, as stated earlier, *DCG's* reported overhead for generating an instruction at run-time is about 350 instructions per instruction generated [Engler, 1994]. It is possible to reduce the cost of run-time code generation by *pre-compiling* as much of the code as possible. Previous researchers have focused on the use of templates, which are sequences of machine instructions containing *holes* in place of some values. Code is generated by copying templates and instantiating the *holes* with values computed at run-time [Lee, 1996]. Until recently, templates were error prone and not very portable.

Now there are automatic derivations of templates. The only problem now is that templates severely limit the range of optimizations that may be applied at run-time. The *Fabius* compiler minimizes the cost of run-time code generation while allowing a wide range of optimizations in both statically and dynamically generated code [Lee, 1996]. The efficient code is generated in a single pass by a relatively simple code generator. No intermediate representation is required at run-time. This approach to some extent does compromise their ability to generate high quality code. For example, it is very difficult to avoid creating jumps to jumps when generating code for conditionals during execution. Other optimizations, such as instruction scheduling are difficult to complete in one pass. An average of 4.7 instructions were required to generate an instruction at run-time which is better than the 350 required by the *DCG* system. The use of *ML* allows the compiler to perform run-time optimizations with little effort on the part of the programmer.

Despite the growing use of dynamic code generation, no mainstream language provides flexible, portable and efficient support for it. Most dynamic code generation systems make the programmer choose between efficiency, ease of programming and debugging, and portability. By generating specialized code for the most active functions, it is possible to gain substantial performance benefits [Hsieh, 1996]. Interpreters can use dynamic code generation technology to improve performance by compiling and then directly executing frequently interpreted pieces of code. *‘C* grew out of the past work with *DCG*. Many improvements were added in *‘C* but the portability and flexibility of *DCG* were retained. The cost of dynamic code generation per generated instruction decreased dramatically from 350 to 10. A high-level interface is provided by *'C* whereas *DCG's* interface is based on the intermediate representation of *lcc* [Fraser, 1991].

Overall, the focus has been to create efficient and portable code generators. Since these are two conflicting interests, a balance must be found between the two or the development of a platform independent language that supports networking protocol characteristics while maintaining a certain level of efficiency. Most researchers have looked toward dynamic code generation for the solution to this problem. This, however, does not necessarily mean there is not a solution using static code generation and protocols.

2.7 BEA Tuxedo® 7.1

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, 1995]. BEA Systems Inc., founded in 1995, is the E-Commerce Transaction Company™, powering many of the world's most innovative e-commerce oriented companies such as Amazon.com, Federal Express, E*Trade, United Airlines, DirectTV and Nokia. The latest version, Tuxedo 7.1, delivers a powerful new security framework for E-Commerce transactions. The security framework allows developers to easily integrate BEA Tuxedo-powered applications with popular third party security software products such as Public Key Infrastructure (PKI) encryption. Digital signatures, digital envelopes and certificate authorities may also be integrated into this framework, thus developing a very high level of security in their e-commerce applications. It is worth noting that security is not one of the focal points for the *MSPL* client-server generation research.

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 also seems more attached to one language than *MSPL*. With the development of another compiler, *MSPL* can easily be used with a new programming language. In Tuxedo, it would be necessary to re-implement the same protocol in the new *target language*. Tuxedo has much more functionality than *MSPL* currently does but most of the features could be added to *MSPL* with more time. The general structure of the message sent between the clients and server, are very similar, almost identical.

The Events-One Way communication method is similar to one command in *MSPL* called *Handshake*, which is explained further in Chapter 3. The general idea in Tuxedo is to allow either the client or server to send a message without receiving a response. The recipient may take some sort of action but does not have to inform the sender about these actions. One such event may be the server informing the client that the server will be unavailable for the next 15 minutes due to maintenance.

The Request/Response communication method is a simple type of dialogue for which the rules are fixed. The client *asks* something and the server *responds*. The client never sends more than one message as part of its request and the server never sends multiple replies to one request. This is the general client-server communication paradigm, which is also used in *MSPL*.

The client-server communication paradigm can be extended to meet the requirements of the third form of communication in Tuxedo, which is Conversational Interactions. This is where the request-response sequence is executed more than once to complete a given service request. It may be necessary in a file transfer when the file being sent is larger than the *buffer* supplied. The server may send 1024 bytes of the file as a reply and then wait for another request from the client saying it is ready for the next 1024 bytes.

The last communication method, Queued Communications, is not implemented by *MSPL*. It is a useful form of communication for when the server is not available for some reason. Some functionality may be lost while the server is down but depending on the role of the server, the client may be able to continue servicing requests and *queue* them to be sent to the server when it is available again.

Another useful feature of Tuxedo is its error handling capability. This is one of the most difficult parts of programming especially in distributed systems. For example, if a request is made and no response is received, there are several probable reasons and solutions to this scenario. One of the reasons may be the server simply did not receive the request or is still processing the request. The request may have even been sent and then lost over the network, thus, the server module is under the impression everything is fine. Sometimes the solution to this problem is not as easy as resending the request. Take for example, if the request was to transfer $1,000,000 from one bank account to another. In this case the programmer wants to be sure to take the correct course of action. Tuxedo uses *transactional communication* to combat this problem. *Transaction communication* ensures each remote operation is done exactly once and all or none of a set of related calls are fulfilled.

BEA Tuxedo supports both library-based and language-based programming. The library-based programming requires programmers to use a set of *C* or *COBOL* procedures defined by BEA Tuxedo. Tuxedo's language-based programming paradigm is a remote procedure call facility called TxRPC, BEA Tuxedo's implementation of X/Open's TxRPC interface [Grenier, 1996].

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 that is portable. By portable it is meant that with additional compilers, there is no need to re-write or re-implement any coding since the entire protocol is encapsulated by *MSPL*.

2.8 Sun's XDR/RPC

Remote procedures calls are defined using an *Interface Definition Language (IDL)*, which contains the definition of the procedure's interface. Communication handling and a binding service are also required. Thus,RPC is a form of distributed communication where the syntax is almost the same as a local procedure call but the called procedure is executed in a different process and usually a different computer from the caller[Coulouris, 1994]. RPC is a simple form of the request/response method discussed in section 2.7, it is modeled after the local procedure call structure. The intent of remote procedure calling is to maintain the semantics of conventional procedure calls in an implementation environment that differs radically. As with local procedure calls, the callers in RPC usually block and wait for the called procedure to complete before regaining control of the CPU. An asynchronous RPC has also been developed and used in distributed window systems such as X-11 [Scheifler, 1986]. The definition of a remote procedure call specifies input and output parameters. Input parameters are the same as parameters *passed by value* in conventional procedure calls. One advantage of RPC, is that by specifying in the IDL, parameters can also be *passed by reference*.

RPC systems developed fall into one of two classes [Coulouris, 1994];

- In the first class, the RPC mechanism is integrated with a particular programming language that includes a notation for defining interfaces.
- In the second class, a special purpose interface definition language is used for describing the interfaces between clients and servers.

Any remote procedure call 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.

As mentioned earlier, there are three main tasks for software that supports remote procedure calling. *Interface processing* involves integrating the RPC mechanism with the client and server programs in conventional programming languages. *Communication handling* is the transmission of request and reply messages using some form of request-reply communication. *Binding* is the process of locating an appropriate server for a particular service.

To build a client program, the RPC system provides a *stub procedure* to stand in for each remote procedure that is called by the client program. For the building of the server program, RPC provides a *despatcher* and a set of *server stub procedures.* The *despatcher* uses the procedure identifier found in the request method to select one of the *server stub procedures* and pass on the arguments. Every procedure in the interface has a unique identifier that is the same on both the client and server sides.

The Sun RPC system provides an interface language called *XDR (External Data Representation)* and an interface compiler called *rpcgen*. Since only one parameter is allowed, procedures requiring more than one must include them as components of a single structure. From the interface definition, the *rpcgen* compiler generates *client stub procedures*, the server main procedure, the *despatcher*, and several *server stub procedures*. Similar to *MSPL*, the application programmer has control over specifying the service port. An extra feature RPC has, is the ability to use UDP (User Datagram Protocol), a connectionless service that transmits messages of up to 64 kilobytes, or TCP connections which is a connection oriented service that transmits streams of bytes across a pre-established connection. *MSPL* currently supports the latter. The level of security offered is not as strong as in BEA Tuxedo but RPC does offer authentication, which may be used with every message sent from the client to the server. The server is then responsible for enforcing access control by deciding whether to execute each procedure call according to the authentication information. The two methods of authentication supported are UNIX and DES (Data Encryption Standard).

Although the RPC is a generally applicable programming mechanism, it seems closely knitted to one language and 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. 9 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*. To the user, the final product is the same in most cases except for the level of efficiency, which is interpreted by the user as the speed of the application or lack thereof. In this section an abstract comparison is made between the two approaches.

Library-based methods provide a fixed list of routines. Consider the purpose of *Java* or *C* versus the development of *Assembly Libraries*. It is possible to code software without the existence of *Java* or *C*. These programming languages provide a programmer with a higher level of abstraction when coding.

Similarly, *MSPL* provides a higher level abstraction for implementing protocols. This has several advantages and disadvantages.

One advantage is the fact that high-level programming languages are easier to read and understand. If this were not the case, then it would be more advantageous to just provide libraries at the *Assembly* level of coding. This would provide more efficient code than most compilers can produce. In fact the degradation in inefficiency due to the use of high-level programming is one of the main reasons why programmers still program in the lower-level *Assembly* for some software packages.

Another advantage is the fact they shorten the development and maintenance time required. This is mainly because the complex low-level communication code is generated by the compiler. Testing of the generated code only has to be done once. After assuring the generated code is error free, it does not have to be tested again after each compilation. 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. This is a great property because any changes in the protocol specification can be done at a higher level, and thus, more easily. The separation also increases portability, as adding compilers for new target languages is relatively easy. 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 meanthat 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. Consider a lock and its matching key, any changes made to the lock must be mirrored with appropriate changes made to the key (or vice versa). Without the mirrored changes, then the key will no longer engage or release the lock. Similarly, 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). In the library-based approach, however, errors on the part of the programmer, may lead to unaligned changes. This 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.

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.

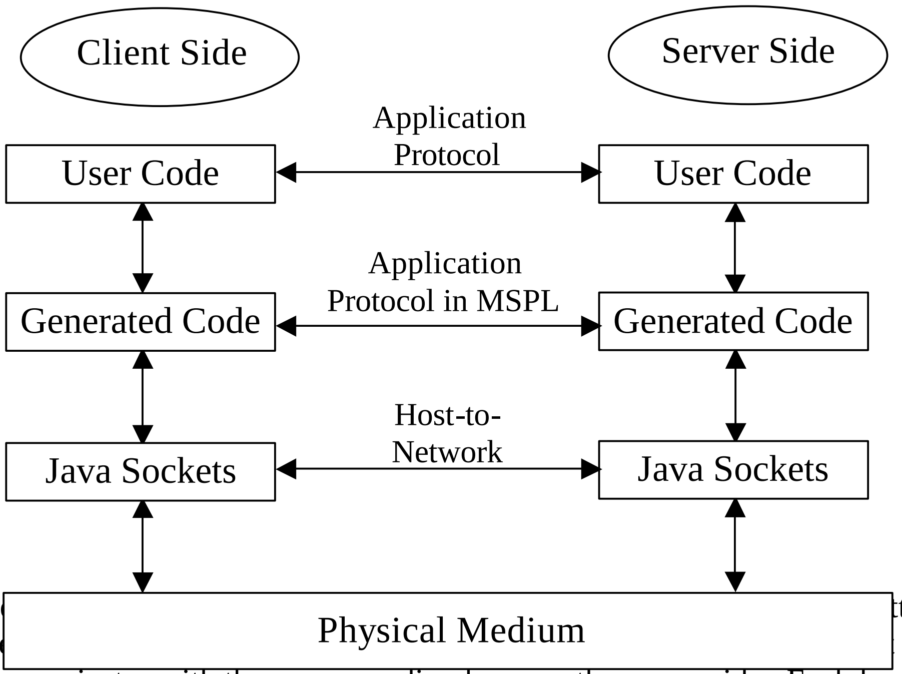
As with the introduction of any other high-level programming language, it is not the answer for all programmers. It does, however, allow more programmers to develop client-server software without having to have an in-depth knowledge of the complex network programming issues that lie underneath. The main drawback is that the more abstract the language, the less efficient the code becomes when compiled into machine code. This depends on how *smart* the compiler is to some extent but not completely.

Chapter 3

MSPL

The first problem stated in the Problem Statement, Section 1.1, 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



Th tted
lin 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

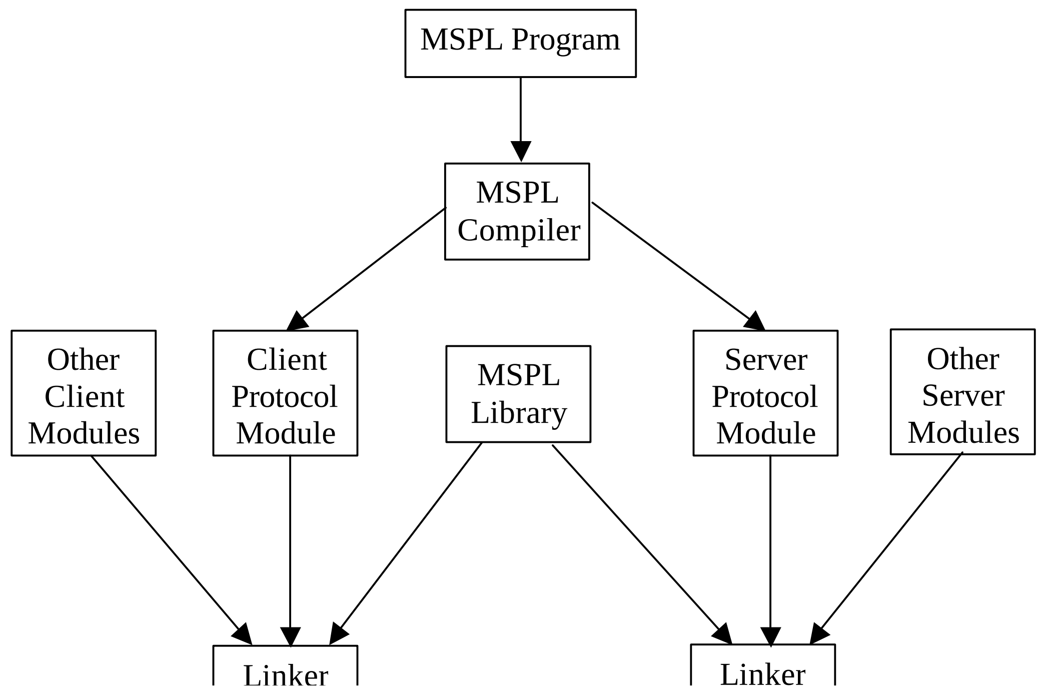
application being developed and outputted by a Compiler with the input of a *MSPL* program.

In all networks, the purpose of each layer is to offer certain services to the higher layers, shielding those layers from the details of how the offered services are actually implemented. In reality, no data is transferred from layer *n* on one machine to layer *n* on another machine. Instead, each layer passes data and control information to the layer immediately below it, until the lowest layer is reached. Below the Java Sockets layer is the Physical Medium through which actual communication occurs. The user-code written by the application programmer, passes data types down to the generated code. The generated client and server protocol modules provide the service of packaging these data types into a Message Packet format and sending it over the network where they are then passed up to the user-written code. The ordering and structure of the messages are specified in *MSPL*.

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 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 chapter. 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. These are the two main differences between this user designed protocol and the standard RFC 959 FTP implementation.

The *ESFTP* application 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. These two machines must also be on the same network. To carry out the function of the application described above, there are three request statements required which are a request to *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 wishing to connect to the server.
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 below:

Put <filename>

1. Client ensures *filename* exists.
2. Client sends command token *Put*, the *filename* and an integer representing the size of the file in bytes.

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 a reply message stating whether the request was completed successfully or not.
5. The client prints the status message to inform the user and then waits for next command/request from the user.

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.

This is the Extremely Simple File Transfer Protocol. It is independent of *MSPL* as it has been implemented by many other programmers without the aid of *MSPL*. The protocol works fine for file transfers, and as its name implies, it is extremely simpler to implement than FTP RFC 959.

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 – 3 the specification-protocol is written for *ESFTP*. This is the same program that generated the portions of code shown in Figure 3 – 8.

Figure 3 – 3. MSPL Code For ESFTP

```
# MSPL file used to generate code for the ESFTP Application
Parameters
   defaultClientPort 55000,  # between 0 and 65535
   defaultServerPort 55000,  # between 0 and 65535
   bufferSize 1000, #same size buffer for Client and Server
   maxClientsSupported 9;
Begin
   Request Get # method for client to receive a file from server
      String Filename;
   Reply Ok
```

```
11.         int statusref,
12.         int length,
13.         byte[] actualFile;
14.      Reply noFile
15.         String noFileError;
16.      Request Put # method for client to send a file to the server
17.         String Filename,
18.         int length,
19.         byte[] actualFile;
20.      Reply Successfull;
21.      Reply fileExists
22.         String overWrite;
23.   End
```

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. Throughout the program, comments may be inserted by preceding the text with a number sign (ie. # this is a comment). The rest of the characters on that line are regarded as a comment and are not processed by the compiler. The default client and server ports may be any integer value from 0 to 65535.

Line 7 signals the beginning of the Request-Reply structure. No parameters can be assigned a value after this keyword. The g*et* 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 10 and 14. 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 *Ok* or *nofile*. If the reply name 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 15 shows, a string follows which may contain more information as to exactly why the request was unsuccessful.

Another possible request is *put*, which is shown on line 16. 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 17, 18 and 19. The two possible replies to this request are *Successful* or *fileExists*. *Successful* is the name of the reply on line 20 and it has no other parameters that are returned with it. 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 21 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.

One other 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 aren't any reply parameters for the *quit* request.

The *quit* command is not written in Figure 3 - 3 because, as mentioned earlier, 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.

In *MSPL*, there are several assumptions that are made in addition to the EBNF definition given in the Appendix. Firstly, a second request cannot be made until a reply for the first request is received unless no reply is expected for the first request. This is critical to maintaining the deterministic order of control, which says a client makes one request and is responded to with one reply.

Secondly, the request parameter timeout is used to re-send or more specifically re-execute one of the automatically generated communication functions. The timeout value is measured in milliseconds. After *n* timeouts, a message is printed to the screen saying the server could not be reached and then control is returned to the application programmer who makes further decisions on the next action.

3.3.1 Definable Communication Parameters

In Figure 3 - 3, the first section of code, between the keywords *Parameters* and *Begin,* is where variables are initialized, giving 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. There is a variable that 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* variable allows you to specify how many clients are allowed to connect to the generated server at any given moment. All the variables have default values incase the programmer doesn't want to set them.

After setting all the parameters desired, then 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 researching 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 chosen to be named *Handshake* in *MSPL*. It is referred to as Events-One Way in Tuxedo.

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 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 compiler is relatively fast since the size of the average *MSPL* program is under 25 lines of code. It takes approximately 4 seconds to generate the *Java* code from the *MSPL* code. Similar to the *Fabius* Compiler described in Chapter 2, the *Java* code is pre-generated with *holes* where values need to be inserted [Lee, 1996]. The current 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.

3.4.1 MSPL Parser

Once the program has been written in *MSPL*, it can be passed onto the Compiler program. The only other input required by the Compiler is the name of the user's server program, which will be called by the generated server code module. Figure 3 - 4 shows a sample run of the Compiler. After entering the required information at the prompt when requested to do so, the code written in *MSPL* is parsed into tokens. The tokens must begin with a letter and are allowed to contain numbers and underscores. Each token is terminated by a white space, semicolon, or comma. Commas and Semi-colons are also considered to be tokens themselves.

Each token is classified as one of the following:

```
Parameters
defaultClientPort
defaultServerPort
bufferSize
maxClientsSupported
Begin
End
Reply
Request
Request_Parameters
timeout
String
int
byte[]
Handshake
Other_Op
Constant_Int
Id
```

Tokens classified as Other_Op may be either a comma or a semicolon.

Figure 3 – 4. Log of Compiler Application Running

```
1.  Script started on Wed Mar 22 19:05:53 2000
2.  CS:1>> java CodeGenerator
3.  Enter Server filename to import: userSMTPD
4.  Enter filename to compile: smtp.mpp

5.  Checking Syntax please wait....
```

```
MySimpleLanguage source code: smtp.mpp
Java generated Server source code: smtpd.java
Java generated Client source code: smtp.java
Generating Code please wait....

Deleting Temporary files...
Deleting Temporary file TEMP/fileOfTokens.mpp...
Successfull!!!
Deleting Temporary file TEMP/temp2000... Successfull!!!

Generated files may be found in GenCode Directory

CS:2>> exit
script done on Wed Mar 22 19:06:11 2000
```

Figure 3 – 5 shows a sample of a file of classified tokens, which is generated as an intermediate step to the final goal of generating the client and server code for the *ESFTP* application. The only possible errors found by the parser are illegal tokens. This means the token contains at least one invalid character.

The names of the server protocol module and client protocol module are derived from the name of the *MSPL* program. The client protocol module is the same name with the ".java" extension instead of ".mpp", while the server protocol module ends with "d.java". After each run of the compiler, all the temporary files created are deleted. These files include the *fileOfTokens.mpp* which is shown in Figure 3 – 5. The file of tokens is created from the *MSPL* code passed to the Compiler. This file is then passed on to the Syntax Table which checks the ordering of the tokens using the Syntax Table shown in Figure 3 - 6.

Figure 3 – 5. Sample Tokens File Generated as Intermediate Step

```
Parameters parameters
defaultClientPort defaultclientport
Constant_Int 25
Other_Op ,
defaultServerPort defaultserverport
Constant_Int 25
Other_Op ,
bufferSize buffersize
Constant_Int 49152
Other_Op ,
maxClientsSupported maxclientssupported
Constant_Int 9
Other_Op ;
Begin begin
Handshake handshake
Other_Op ;
Request request
Id mail
byte[] byte[]
Id id
Other_Op ;
Reply reply
Id ok
byte[] byte[]
Id actualfile
Other_Op ;
Reply reply
Id notokmail
byte[] byte[]
Id notok
Other_Op ;
Request request
Id rcpt
byte[] byte[]
Id toaddress
Other_Op ;
Reply reply
Id okrcpt
byte[] byte[]
Id ok
Other_Op ;
Reply reply
Id notokrcpt
byte[] byte[]
Id notok
Other_Op ;
Request request
Id data
byte[] byte[]
Id sendmessage
Other_Op ;
Reply reply
```

```
Id success
byte[] byte[]
Id successbytes
Other_Op ;
Reply reply
Id failure
byte[] byte[]
Id failurebytes
Other_Op ;
Request request
Id message
byte[] byte[]
Id actualmessage
Other_Op ;
Reply reply
Id messageaccepted
byte[] byte[]
Id successbytes
Other_Op ;
Reply reply
Id messagedenied
byte[] byte[]
Id failurebytes
Other_Op ;
End end
```

The method that classifies tokens ignores comments by discarding them since they are not needed for compilation of the code.

3.4.2 Checking Syntax of MSPL Code

After parsing the file into tokens and classifying each token, then the actual syntax is checked. This is the process where most errors are found. By this time we are assured the file being compiled exists and contains all legal tokens. Now we may look at the ordering of these tokens to determine if we can generate code from them. The entire syntax of *MSPL* has been placed in a table called the *MSPL Syntax Table* shown in Figure 3 – 6.

Figure 3 - 6. MSPL Syntax Table

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	Parameters	50	1	2	3	4	51	50	50	50	50	50	50	50	50	50	50	50	50	50
1	defaultClientPort	59	52	52	52	52	52	52	52	52	52	10	52	52	52	52	52	52	52	52
2	defaultServerPort	59	52	52	52	52	52	52	52	52	52	10	52	52	52	52	52	52	52	52
3	bufferSize	59	52	52	52	52	52	52	52	52	52	10	52	52	52	52	52	52	52	52
4	maxClientsSupported	59	52	52	52	52	52	52	52	52	52	10	52	52	52	52	52	52	52	52
5	Begin	59	53	53	53	53	53	6	53	53	53	53	53	53	53	53	53	53	53	18
6	Request	59	54	54	54	54	54	54	7	54	54	54	54	54	54	54	54	54	54	54
7	ID	59	55	55	55	55	55	55	55	55	55	55	55	12	13	14	15	16	55	55
8	Request_Parameters	59	56	56	56	56	56	56	56	56	9	56	51	56	56	56	56	56	56	56
9	timeout	59	52	52	52	52	52	52	52	52	52	10	52	52	52	52	52	52	52	52
10	Constant_Int	59	57	57	57	57	57	57	57	57	57	57	57	57	57	57	15	16	57	57
11	Reply	59	58	58	58	58	58	58	7	58	58	58	58	58	58	58	58	58	58	58
12	string	59	58	58	58	58	58	58	7	58	58	58	58	58	58	58	58	58	58	58
13	Int	59	58	58	58	58	58	58	7	58	58	58	58	58	58	58	58	58	58	58
14	Byte[]	59	58	58	58	58	58	58	7	58	58	58	58	58	58	58	58	58	58	58
15	,	59	1	2	3	4	60	60	61	60	9	62	60	12	13	14	63	64	60	74
16	;	59	65	65	65	65	5	6	61	8	65	62	11	65	65	65	66	67	17	74
17	End	59	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68
18	Handshake	59	57	57	57	57	57	57	57	57	57	57	57	57	57	57	57	16	57	57

The rows represent every possible token that is accepted in *MSPL* and the error states that may be entered depending on the next token input. For example, if the current state is *Parameters* and the next token input is anything other than a parameter variable (ie. defaultClientPort, bufferSize*etc.*) you will go to an error state. The error states are 50 and greater. States 19 to 49 are reserved for extending the language. The error states are not shown in the table but each number greater than 50 refers to an error message, which is printed when that state is entered. The number of error states is large to enable more specific error messages to be printed to the screen. If there was only one error message, it would have to be very general such as “Error Found”. By increasing the number of error messages, each message can be more specific to the problem encountered.

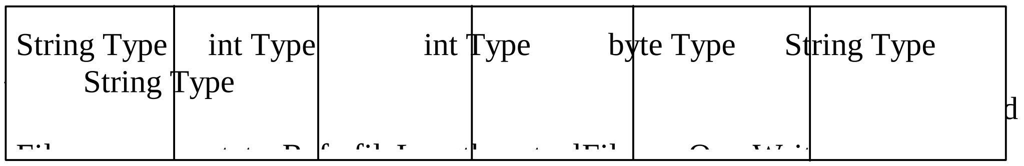
3.5 Generated Protocol Modules

Once the program written in *MSPL* passes through the Syntax Process 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 client file, the server file, the server interface file and the message packet 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. A graphical representation of the *ESFTP* message packet is shown in Figure 3 – 7.

Figure 3 – 7. ESFTP Message Packet Structure



in Figure 3 - 7 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 – 7. 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 checkinga standard variable called the Reply name. This is a part of the *MSPL* protocol.

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 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*.

An 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- 8 shows the interface class for the *ESFTP* example described throughout Chapter 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. This is called the Application Protocol and is the responsibility of the application programmer to specify.

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 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 g*et* 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.

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

Figure 3 – 8 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 compiler.

The *put* method is shown in Figure 3 – 8. 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.

Figure 3 – 8. Sample-Generated and User-Written Code for ESFTP

Client

```
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;
}
```

Server Protocol Module

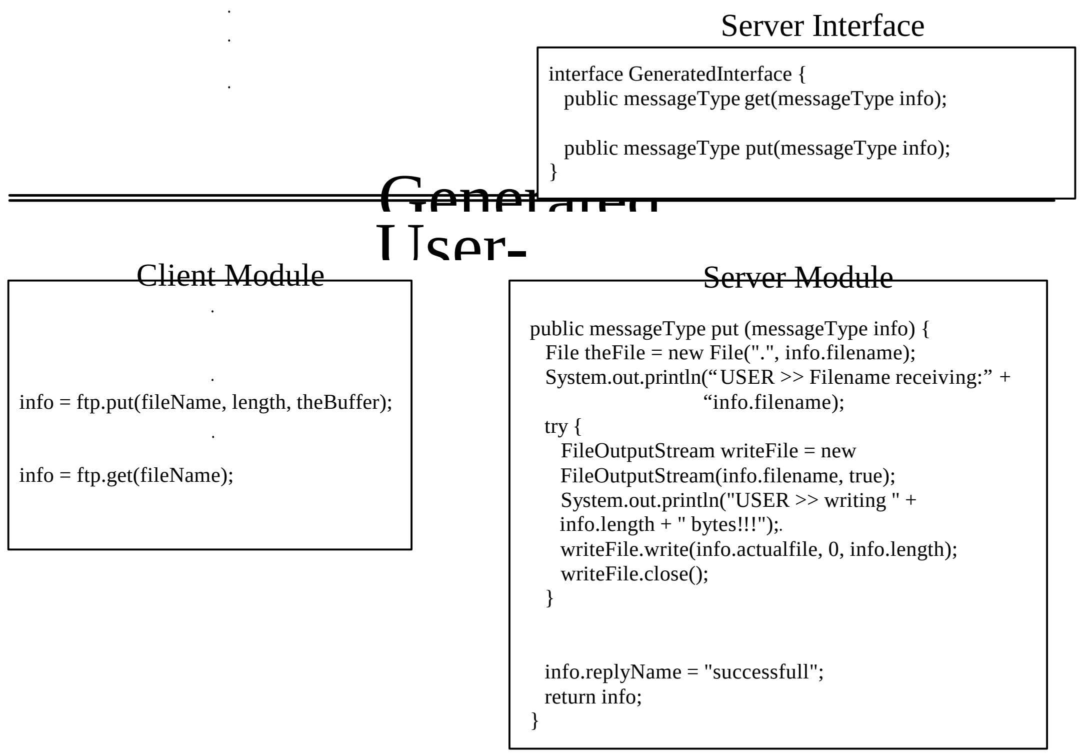
```
stringLength = receive.readInt();

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);
```



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

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

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

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

```
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;
}
```

3.8 MSPL Library

There are several advantages to developing a library that is linked to the generated code. They are great to help increase portability but the arrival of new hardware platforms requires the redesign or at least extensive re-coding of the libraries in general. Currently, the *MSPL Library* is not that extensive. Most of the code linked to the generated modules and the user-written modules are found in standard java packages. A potential use for the library in future could be to add any RFC specific modules that are standard. For example, there may be one or two modules used by FTP RFC 959 that could be added to the *MSPL Library*. This may allow more control over the code being generated, and thus lead to more efficient and reliable generated code.

Chapter 4

Implementation of RFC Protocols

In this chapter, we take a closer look at how *MSPL* can be used to implement *Real World* protocols. As experiments for proof of concept and usability, 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)*. After compiling the *MSPL* programs, sample user code was also written. Communication between the generated client code and another existing server which implements the same RFC was attempted as was communication between the generated server code with other existing clients that implement the same RFC.

The goal of testing a generated client with an existing server and a generated server with an existing client is to demonstrate that '*real world*' protocols can be specified in *MSPL* and the Compiler produces the appropriate code for communication.

4.1 Implementation of SMTP RFC 821

The Simple Mail Transfer Protocol is used for the sending and receiving of electronic mail. It is independent of the particular transmission subsystem and requires only a reliable ordered data stream channel. The general model of communication is that as the result of a user mail request, the sender-SMTP establishes a two-way transmission channel to a receiver-SMTP. The receiver-SMTP may be either the ultimate destination or an intermediate. SMTP commands are generated by the sender-SMTP and sent to the receiver-SMTP. SMTP replies are sent from the receiver-SMTP to the sender-SMTP in response to the commands [Postel, 1982].

In the SMTP example, the code written in *MSPL* is shown in Figure 4 – 1. The generated client along with user's client code, was used to send a message through "winnie.fit.edu ESMTP Sendmail 8.9.3/8.9.1" server. This was done successfully and Figure 4 – 2 is a script of the communication that occurred between the generated client and the Florida Tech ESMTP server.

The defaultClientPort and the defaultServerPort on lines 2 and 3 were both set to 25, which is the standard port to communicate on for SMTP. The *buffersize* was set to1024 bytes on line 4. Since only the client side was implemented the maxClientsSupported on line 5 did not play a major role in the script shown in Figure 4 – 2. The *Handshake* command on line 7 is required for the RFC 821 implementation of SMTP. To successfully send a message, four requests had to be implemented. These requests were called *MAIL*, *RCPT*, *DATA* and *MESSAGE* and can be found on lines 8, 14, 20 and 26 respectively. All of these requests as for most RFC protocols, require a stream of bytes to be sent between the client and server, therefore, the field added by each of these requests is a variable that stores bytes.

Figure 4 – 1. MSPL Code for SMTP

```
Parameters
   defaultClientPort 25,  # between 0 and 65535
   defaultServerPort 25,  # between 0 and 65535
   bufferSize 1024,  # same size buffer for Client and Server
   maxClientsSupported 9;

Begin
   Handshake;  # used to inform client of version of server
   Request mail  # command to identify sender of message
      byte[] id;
   Reply mailOk  # accept sender address
      byte[] actualFile;
   Reply mailError # reject sender address
      byte[] error;

   Request RCPT  # address of potential mail recipient
      byte[] toaddress;
   Reply rcptOk  # accept recipient address
      byte[] ok;
   Reply rcptError # reject recipient address
      byte[] error;

   Request DATA  # Prepare to send text message
      byte[] sendmessage;
   Reply dataOk  # text message sent successfully
      byte[] dataOkBytes;
   Reply dataError #unable to send message
      byte[] failurebytes;

   Request message  # command to send actual message
      byte[] actualmessage;
   Reply messageaccepted   # message sent successfully
      byte[] successbytes;
   Reply messagedenied  # message rejected for some reason
      byte[] failurebytes;
   End
```

As mentioned earlier, the *quit* command does not have to be specified in the *MSPL* program because it is standard over most protocols. Therefore, the *quit* command is generated automatically. In most if not all the RFC protocols, there are two parts to a reply message. The first part is usually a number that specifies the type of reply being sent and the second part is a string, which helps the user understand which reply is being sent. This is the format you will observe in all the conversation scripts through out this chapter.

Figure 4 – 2. SMTP Conversation Script

```
Script started on Wed Mar 22 02:34:39 2000
CS:1>> java userSMTP
Starting userSMTP application...
```

```
Enter Address to Connect To >> fit.edu
From Server: 220 winnie.fit.edu ESMTP Sendmail 8.9.3/8.9.1; Wed, 22
   Mar 2000 02:34:39 -0500 (EST)

UserSMTP>> MAIL FROM:Melvin@research.com
Mail command:
            'MAIL FROM:Melvin@research.com'
Executing mail function
Finished sending Request Parameters
Returning control to user...

From Server: 250 Melvin@research.com... Sende r ok
   3/8.9.1; Wed, 22 Mar 2000 02:35:24 -0500 (EST)
User >> Finished executing MAIL function!!!

UserSMTP>> RCPT TO:mdouglas@fit.edu
Recipient command:
            'RCPT TO:mdouglas@fit.edu'
Executing rcpt function
Finished sending Request Parameters
Returning control to user...

From Server: 250 mdouglas@fit.edu... Recipient ok
   3/8.9.1; Wed, 22 Mar 2000 02:35:37 -0500 (EST)
User >> Finished executing RCPT function!!!

UserSMTP>> DATA
DATA command: DATA
Executing data function
Finished sending Request Parameters
Returning control to user...

From Server: 354 Enter mail, end with "." on a line by itself, 22
   Mar 2000 02:35:38 -0500 (EST)

*********
Message Text:
            Hello,
               This is a message being sent from Melvin's Generated
               SMTP Client.  It is interacting with the Florida
               Tech ESMTP server.
            .

Executing message function
Sending bytes in: actualmessage
Finished sending Request Parameters
Returning control to user...

From Server: 250 CAA07090 Message accepted for delivery
   , 22 Mar 2000 02:37:13 -0500 (EST)

TEXT COMPLETE
   *********
User >> Finished executing DATA function!!!
```

```
UserSMTP>> **quit**
UserSMTP>> Quiting FTP Application
Finished sending Request Parameters
Returning control to user...

From Server: 221 winnie.fit.edu closing connection
   , 22 Mar 2000 02:37:25 -0500 (EST)

UserFTP>> Thank for using this code Generated FTP Application
User >> Finished executing quit function!!!
CS:2>>
   CS:2>> exit
script done on Wed Mar 22 02:37:36 2000
```

Line 2 shows how the client software is started. Currently it is text based *Java* program. With the information supplied on line 4, the*connectTo* method in the generated client module, establishes a connection with the fit.edu ESMTP Sendmail server. Line 5 shows the *Handshake* sent by the server stating the version of software running on the server and response code 220 signifying it is ready to service requests from the client. The first request from the client is to inform the server of the sender address shown on line 6. The format for this request is shown on the next line;

MAIL <SP> FROM:<reverse-path> <CRLF>

The server can accept or deny the sender address. In this conversation, line 11 shows the server acknowledges the address Melvin@research.com as being *ok* with response code 250. The next request is to inform the server of where the message should be sent. This is known as the recipient address and the format for this

request message is shown on the following line;

RCPT <SP> TO:<forward-path> <CRLF>

This request is made on line 13 of the SMTP conversation script to make mdouglas@fit.edu a recipient. The server then responds to this message on line 18, again with a response code of 250. To add a second recipient, the request on line 13 would be repeated but with the different address desired. The *Data* request is now a possible option as a request by the client. Without the previous information of recipient and sender address, this request would not be accepted. The *Data* request shown on line 20, tells the server to prepare to receive the actual text message. The format for this request is shown below;

DATA <CRLF>

If the server is capable of receiving the text message right away, it sends a response code of 354 as shown on line 25. Response code 354 means enter a message and end it with a "." on a line by itself. Once the user enters a text message as shown on line 27, then it is sent and the server acknowledges whether the message was accepted or not as shown on line 32. At this point the user may choose to send another message or to end their session. To end the session the *quit* request is used as shown on line 35. The server replies on line 39 with response code 221, which means "service closing transmission channel" [Postel, 1982].

4.2 Implementation of HTTP RFC 2616

HTTP has been in use by the World-Wide Web global information initiative since 1990 [Fielding, 1999]. The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol that can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems. This would be done through extension of its request methods, error codes and headers [Masinter, 1998].

The *HTTP* protocol is a request/response 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 responds with a status line, including the message's protocol version and a success or error code, followed by several lines with server information [Fielding, 1999]. HTTP communication usually takes place over TCP/IP connections. The default port is 80, but other ports can be used [Reynolds, 1994].

For the purpose of a functional client and or server, the only methods required were the *GET* and *QUIT* methods. Figure 4 – 3 shows the *MSPL* code used to generate the java protocol modules.

Figure 4 – 3. MSPL Code for HTTP

```
Parameters
   defaultClientPort 55000,  # between 0 and 65535
   defaultServerPort 55000,  # between 0 and 65535
   bufferSize 1024,  # same size buffer for Client and Server
   maxClientsSupported 10;

Begin
   Request Get;
   Reply Ok # successfully received, understood, and accepted
      byte[] actualFile;
     Reply fileNotFound        # The request contained bad
        byte[] errorFourHundred; # syntax or cannot be
fulfilled
```

```
12.        Reply serverNotAvailable  # The server failed to fulfill
13.           byte[] errorFiveHundred; # an apparently valid
      request
14.         End
```

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 seems to work moderately 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.

4.2.1 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– 4. The lines in bold are the relevant pieces of information that are currently being used to service the requests. It is possible, however, to increase the functionality of the server by using more of the information provided to the server from the client.

Figure 4 – 4. HTTP Server Conversation Script

```
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
4.     Document Root: .
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
18.    Bytes Sent: 2279
19.    Filelength: 2279
20.    USER HTTPD >> Finished Sending File!!!
21.    End Request
22.    Now sending reply for request just received...
23.    Sending bytes
```

24. End Request-Reply...

25. Request: get
26. USER HTTPD >> *** BEGIN HTTP Packet:
27. **GET /Images/Flagbda.gif HTTP/1.1**

28. Accept: */*

29. **Referer: http://fit.edu:55000/~mdouglas**

30. Accept-Language: en-us

31. Accept-Encoding: gzip, deflate

32. User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows NT; DigExt)

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
38. Bytes Sent: 33325
39. Filelength: 33325
40. USER HTTPD >> Finished Sending File!!!
41. End Request
42. Now sending reply for request just received...
43. Sending bytes
44. End Request-Reply...

45. Request: get
46. USER HTTPD >> *** BEGIN HTTP Packet:
47. **GET /Images/Whatsnew.gif HTTP/1.1**

48. Accept: */*

49. **Referer: http://fit.edu:55000/~mdouglas**

50. Accept-Language: en-us

51. Accept-Encoding: gzip, deflate

52. User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows NT; DigExt)

53. **Host: fit.edu:55000**

54. Connection: Keep-Alive
55. USER HTTPD >> *** END HTTP Packet.

```
56.    USER HTTPD >> pathname: '/Images/Whatsnew.gif'
57.    USER >> Filename sending:Images/Whatsnew.gif
58.    Bytes Sent: 17709
59.    Filelength: 17709
60.    USER HTTPD >> Finished Sending File!!!
61.    End Request
62.    Now sending reply for request just received...
63.    Sending bytes
64.    End Request-Reply...
65.    ^C
       /usr/users/student/mdouglas/public_html> exit
66.    script done on Wed Mar 29 01:58:45 2000
```

Line 2 in Figure 4 – 4 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, 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, 1999].

A second web page is requested on line 26. This file is referred to by a link on the current page, therefore, line 28 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 shown on the following line;

Referrer = "Referrer" ":" (absoluteURI | relativeURI)

In HTTP/1.0, 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, 33 and 53 inform the server of whether the connection should be closed or not. A Connection, however, may be closed for a variety of other reasons.

There are several advantages to having one persistent HTTP connection instead of several separate TCP connections. Firstly, latency on subsequent requests is reduced since there is no time spent in TCP's connection opening handshake. Secondly, network congestion is reduced by reducing the number of packets required for TCP opens [Fielding, 1999].

After receiving and interpreting a request message, the server responds with an HTTP response message for which the protocol is shown below;

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 can be seen on the following line;

Status-Line = HTTP-Version <SP> Status-Code <SP> Reason-Phrase <CRLF>

Figure 4 – 5 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.

Figure 4 – 5. HTTP Status Codes [Fielding, 1999]

- 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.

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.

4.2.2 HTTP Client Software

For HTTP RFC 2616, the client was also generated. In Figure 4 – 6, 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.

Figure 4 – 6. HTTP Client Conversation Script

```
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 – 6 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 file they would like to *browse*. Lines 6 to 13 inclusive, 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 shown on the following line;

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, as in previous protocols, 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.

4.3 Implementation of FTP RFC 959

The File Transfer Protocol RFC 959 is used to transfer data reliably and efficiently between two machines. It shields a user from variations in file storage systems among hosts [Reynolds, 1985]. One machine must have the server running while the second machine makes requests through a FTP client.

The general File Transfer Protocol model is similar to ESFTP, as described in Section 3.2 earlier. The main differences, which were also mentioned earlier are the handshake that is sent initially and the second *data connection* used to transfer actual bytes of files. The communication between the user and server is intended to be an alternating dialogue. Certain commands require a second reply for which the user should also wait. These replies for example, may report the closure of the data connection.

During the implementation of the FTP RFC 959 protocol, several interesting problems arose. Some were overcome, while others were too in-depth and have been left as interesting prospects for future work.

The first problem encountered was that many protocols started the conversation with the server sending a message first, basically informing the client it was ready to communicate using a specific version of the application. This problem was overcome by extending *MSPL* to include the *Handshake* command, which allows the server to send a stream of bytes for which there is no Reply, hence, it was not defined as a Request but as a *Handshake* command. Since the syntax of the language was encoded into a table format, this extension was not that hard to do.

The second problem encountered was that in RFC 959, FTP has two data connections in operation at any given moment. One connection is called the *control connection* and the second is called the *data connection* [Reynolds, 1985]. Over the *control connection*, request for services are made. If the reply involves sending or receiving a file or a list of all the files in the current directory, then a second connection called the *data connection* is opened. This connection remains open only long enough to fulfill the Request made, and then it is closed.

There are two reasons why this was a problem that could not be solved using the current version of *MSPL*. The first reason is that the current version does not support more than one connection between the client and server. The second reason was that the assignment of the connection on the second port would have to be dynamic, changing during one execution of the program. Possible solutions to this problem are discussed in Chapter 5. Figure 4– 7 shows the *MSPL* code written that works successfully for the commands that do not require a second *data connection*. Some of these commands are print working directory (*pwd)* and *change directory (cd).*

Figure 4 – 7. MSPL Code for FTP

```
Parameters
   defaultClientPort 55000,  # between 0 and 65535
   defaultServerPort 55000,  # between 0 and 65535
   bufferSize 1024, # buffersize in bytes
   maxClientsSupported 9;

Begin
   Handshake;
   Request user;
   Reply goodusername
      byte[] needPassword;
   Reply badusername
      byte[] stop;
   Request pass;
   Reply goodpassword
      byte[] ready;
   Reply badpassword
      byte[] stop;
   Request get;
   Reply gettwohundred # positive completion reply
      byte[] getgood;
   Reply getFourHundred #temporary negative reply-try later
      byte[] msgFourHundred;
   Reply getFiveHundred # permanent negative reply
      byte[] msgFiveHundred;
   Request pwd;
   Reply currentpath  # positive completion reply
      byte[] msgTwoHundred;
   Reply errorPWD
      byte[] error;
   Request cwd;
   Reply cwdOk
      byte[] msgTwoHundred;
   Reply cwdError
      byte[] error;
End
```

The standard port used for FTP communication is port 21. To avoid having to shut down the current FTP server running on this port, the *MSPL* code set up the server to listen on port 55000 by assigning this value to the *defaultClientPort* and *defaultServerPort* on lines 2 and 3. Since no files were able to be transferred, the specified *buffersize* of 1024 was more than sufficient to transfer any one request or reply in its entirety. The use of the *Handshake* command on line 7 was also necessary for this protocol. Several Request–Reply statements were coded in *MSPL* for this example. The request *user* on line 8 is a request for the server to log the client on with the specified username that follows the method name *user*. The *Handshake* command is where the server asks the client to provide this information. If the username is valid then the server sends back an integer signifying that a password is also required. Once the password has been confirmed valid then the client is free to make other requests such as *cd* or *pwd*. The implementation of the other commands using the second *data connection* has been left as future work. The script for the generated server is shown in Figure 4 - 8. It shows the messages that are exchanged with the standard ftp client on the winnie.fit.edu server.

Figure 4 – 8. FTP Conversation Script

```
Script started on Wed Mar 29 15:38:58 2000
CS:1>> java ftpd
```

```
3.     Accepting connections on port 55000
4.     Document Root: .

5.     Connection Established!!!

6.     Handshake Sent…
7.     Now receiving parameters for Request: user
8.     Server got username: 'USER Melvin'
9.     Finished receiving request parameters called user module
       now send reply info
10.    Now sending reply for request just received...
11.    Sending bytes
12.    Finished executing command...Waiting for next command...

13.    Now receiving parameters for Request: pass
14.    Server got password: 'PASS SonOfTheMostHigh'
15.    Finished receiving request parameters called user module
       now send reply info
16.    Now sending reply for request just received...
17.    Sending bytes
18.    Finished executing command...Waiting for next command...

19.    Now receiving parameters for Request: pwd
20.    Finished receiving request parameters called user module
       now send reply info
21.    Now sending reply for request just received...
22.    Finished executing command...Waiting for next command...
23.    ^C

24.    CS:2>> exit

25.    script done on Wed Mar 29 15:40:34 2000
```

The *pwd* command shows the current directory path on the server machine. The path is sent back to the client. An example of the path sent back would be *export/home/gsa/mdouglas.* This type of request does not require a second data connection and thus, can be done using the current version of *MSPL*.

The first five lines of the FTP Conversation Script have the same purpose as those in the HTTP Conversation Script. Once a connection has been established, however, a *Handshake* is sent from the server to the client in which the client is informed of the version of software being run and whether the server is ready to service requests. Line 7 shows the server receiving the client’s request for a user to log on to the server with the specified username.   The FTP command sent by the client side is shown below;

```
USER <SP> <username> <CRLF>
```

It is now up to the application protocol to decide if this is a valid username it wants to accept. In Figure 4 – 8, the application protocol decides the username is fine but a password is also required. Therefore, the reply sent signifies username was accepted but a password is needed. The client application protocol understands this response and sends the user’s password as shown below;

PASS <SP> <password> <CRLF>

The request is received by the server on line 13. The server module accepts the password as being valid and then sends this information back to the client protocol module, which then passes it on to the client module.

One of the few commands that do not require the use of the *data connection* shown in Figure 4 – 10, is the *pwd* command. This command sends the Reply back over the FTP Replies line shown in Figure 4 – 10. The FTP Replies line is the same connection that the client sends the FTP commands or requests over. The command is used as shown on the next line;

```
PWD  <CRLF>
```

FTP commands are "Telnet strings" terminated by the "Telnet end of line code". The command codes themselves are alphabetic characters terminated by the character <SP> (Space) if parameters follow and Telnet-EOL otherwise [Reynolds, 1985]. Similar to HTTP, there are five possible types of reply codes shown in Figure 4 - 9.

Figure 4 – 9. FTP Status Codes [Reynolds, 1985]

- 1yz Positive Preliminary reply

  The requested action is being initiated; expect another reply before proceeding with a new command.

- 2yz Positive Completion reply

  The requested action has been successfully completed. A new request may be initiated.

- 3yz Positive Intermediate reply

  The command has been accepted, but the requested action is being held in abeyance, pending receipt of further information. The user should send another command specifying this information. This reply is usd in command sequence groups.

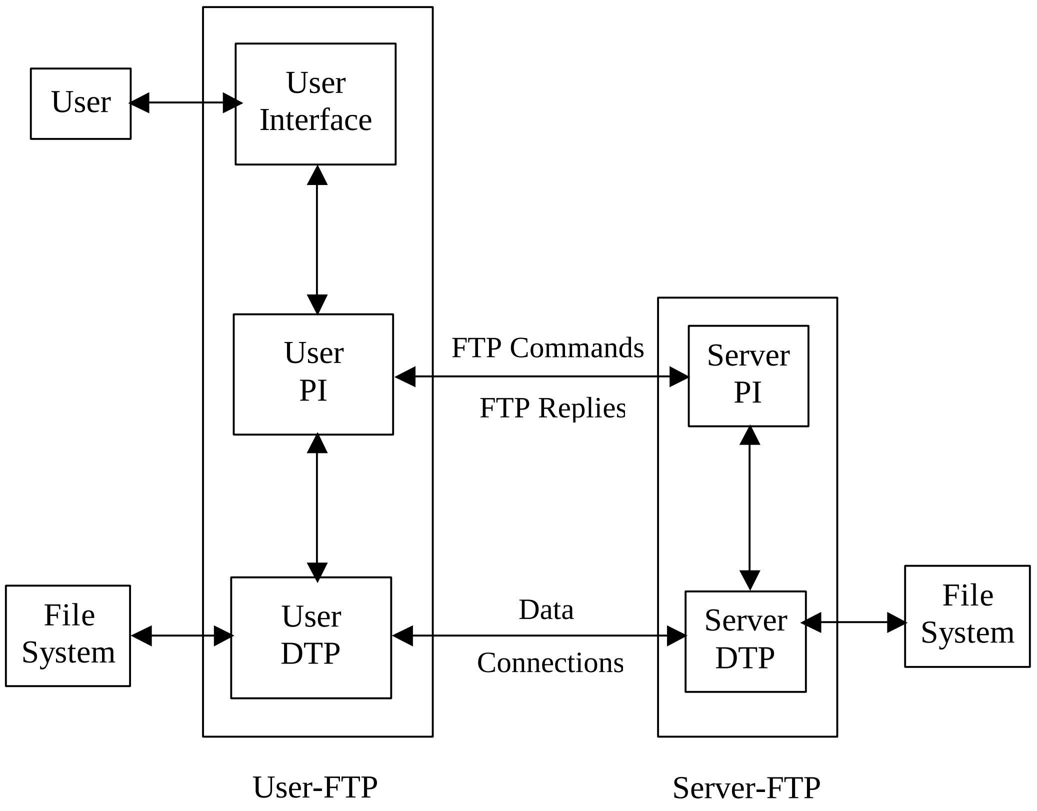
- 4yz Transient Negative Completion reply

The command was not accepted and the requested action did not take place, but the error condition is temporary and the action may be requested again. The user should return to the beginning of the command sequence, if any.

- 5yz Permanent Negative Completion reply

  The command was not accepted and the requested action did not take place. The User-process is discouraged from repeating the exact request (in the same sequence).

Figure 4 – 10. FTP Model [Reynolds, 1985]



In Figure 4 – 9, there is a reference to User DTP (Data Transfer Process) and Server DTP. There is also mention of a User PI (Protocol Interpreter) and Server PI which is the same as generated Client and Server Protocol modules. The User and Server DTP are the two modules that would have to be developed in order to use the *put* and *get* commands among other similar commands that require the use of the *data connection*. Chapter 5 discusses possible extensions in *MSPL* and the Compiler in order to generate a User DTP and Server DTP.

Chapter 5

Conclusion

Overall, the research done can be considered a success. There have been several definitive steps taken in the right direction to increase the level of quality in the development of client-server software. The Compiler and *MSPL* combined, 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.

5.1 Significance and Expected Impact of Research

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 just 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 and thus, not much evidence to base this statement on. The independent development of *MSPL* from the compiler makes this solution very portable since a compiler from *MSPL* code to any programming language can easily be developed.

5.2 Prospects for Future Work

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*. Other future works include improving the compiler. By making it smarter, it can optimize the *MSPL* code before compilation.

Another interesting future work would be to provide more error handling features similar to the BEA Tuxedo package described in Chapter 2. This would greatly increase the reliability of the language when used in the *Real World.*

Appendix

EBNF definition for MSPL

```
<My_Simple_Language>    ::= <declaration> <body>
<declaration>           ::= Parameters <global_parameter_list> | ?
<global_parameter_list> ::= <global_parameter> <term> |
                              <global_parameter> <term>,
                              <global_parameter_list>
<global_parameter>      ::= defaultClientPort | defaultServerPort |
                              bufferSize | maxClientsSupported
<term>                  ::= Constant_Int
<body>                  ::= Begin <statement_list> End |
                              Begin Handshake; <statement_list> End
<statement_list>        ::= <statement> |
                              <statement> <statement_list>
<statement>             ::= <request_statement> <reply_statement>
<request_statement>     ::= Request id <var_list> |
                              Request id <var_list>
                              Request_Parameters <parameter_list>
<var_list>              ::= <type> id; | <type> id, <var_list>
<type>                      ::= int | String | byte[]
<reply_statement>       ::= ? | Reply id <var_list> |
                              Reply id <var_list> <reply_statement>
<parameter_list>        ::= <parameter> <term> | <parameter>
                              <term>, <parameter_list>
<parameter>             ::= timeout
```

References

[Amarasinghe, 1993] S. P. Amarasinghe and M. S. Lam, “Communication Optimization and Code Generation for Distributed Memory Machines,” *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, June, 1993. http://researchsmp2.cc.vt.edu/DB/db/conf/pldi/pldi93.html

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

[Clark, 1990] D. D. Clark and D. L. Tennenhouse. “Architectural Considerations for a New Generation of Protocols,” ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1990, September 1990.

[Coulouris, 1994] George Coulouris, Jean Dollimore, Tim Kindberg,“Distributed Systems Concepts and Design,” pp. 130 - 152 Second Edition 1994.

[Engelen, 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/

[Engler, 1994] Dawson R. Engler and Todd A. Proebsting, “DCG: An Efficient, Retargetable Dynamic Code Generator,” *ASPLOS-VI Proceedings - Sixth*

International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, October 4-7, 1994. http://www.stanford.edu/~engler/

[Fielding, 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.

[Fraser, 1991] Chritopher W. Fraser and David R. Hanson. "A Code Generation Interface for ANSI C. Software-Practice and Experience," 21(9):963-988, Proceedings of the International Workshop on Code Generation, September 1991.

[Grenier,1996] Christina Grenier, "The TXRPC Specification, X/Open CAE Specification," November 1995.

[Hsieh, 1996] Dawson R. Engler and M. Frans Kaashoek, "`C: A Language for High-Level, Efficient, and Machine-independent Dynamic Code Generation," *The 23rd annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida on January 21-24, 1996. http://www.stanford.edu/~engler/

[Kohler, 1999] E. Kohler, M. F. Kaashoek, and D. R. Montgomery (MIT) "A Readable TCP in the Protocol Language," *SIGCOMM 1999*, August 1999.

[Lee, 1996] Peter Lee, Mark Leone, "Optimizing ML with Run-Time code generation," *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadelphia*, Pennsylvania, May 21-24, 1996. SIGPLAN Notices 31(5), May 1996.

http://www.acm.org/sigmod/dblp/db/conf/pldi/pldi96.html

[Masinter, 1998] L. Masinter, “Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0),” RFC 2324, April 1998.

[Postel, 1982] Jonathan B. Postel, “Simple Mail Transfer Protocol RFC 821”, August 1982.

[Reynolds, 1985] J. Reynolds, J. Postel, “File Transfer Protocol RFC 959,” October 1985.

[Reynolds, 1994] Reynolds, J. and J. Postel, “ Assigned Numbers”, STD 2, RFC 1700, October 1994.

[TanenBaum] **Andrew S. TanenBaum, “Computer Networks” Third Edition, pp. 28-29, 1996.**

[Scheifler, 1986] R.W. Scheifler, J. Gettys, “The X Window System” ACM Trans. On Computer Graphics, pp. 76-109, 1986.