Active Library Resolution in Active Networks

David C. Lee

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Electrical Engineering

Scott F. Midkiff, Chair
James D. Arthur
Nathaniel J. Davis, IV
F. Gail Gray
Charles E. Nunnally

February 20, 1998
Blacksburg, Virginia

Keywords: Library Resolution, Active Networks
Copyright © 1998, David C. Lee

Active Library Resolution in Active Networks

David C. Lee

(ABSTRACT)

An active network is a computer network in which new protocols can be installed at run-time in any node within the network. For example, the deployment of Internet multicast technology has been slow because service providers have been reluctant to upgrade and reconfigure their routing nodes. Under the active network scheme, users who desire multicast services can have the service automatically installed without any direct intervention by the user or the provider.

One major question in realizing active networks is how the code for the new active library can be found, or resolved, and retrieved. A model of the resolution and retrieval mechanisms is the major focus of this research. To validate the model, a proof-of-concept experimental system that realizes a simple active network architecture was developed. An active library resolution service model, suitable for a global Internet, was investigated using this experimental platform and a simulation system. The two protocol components that were built and studied are the active transport protocol and the active library resolution protocol.

The experimental and simulation systems were used to evaluate the extensibility, overhead, resolution time, scalability, and policy constraint support of the service. Extensibility and policy constraint support are an integral part of the proposed design. For libraries located on servers that are at most ten hops away from the requesting source, the resolution time is under 2.6 seconds. Simulations of networks of different sizes and with different error rates exhibit linear resolution time and overhead characteristics, which indicates potential scalability. Behavior under high loss rates showed better than expected performance. The results indicate that the library resolution concept is feasible and that the proposed strategy is a good solution.

Acknowledgments

The members of my committee, Dr. J.D. Arthur, Dr. N.J. Davis, IV, Dr. F.G. Gray, Dr. C.E. Nunnally, and Dr. S.F. Midkiff, have provided valuable guidance and constructive feedback on this work. I truly appreciate their involvement and also value the excellent advice, enlightenment, and continuous support provided by my advisor, Dr. Midkiff.

Thanks to my family, Kun-chieh, Shiow Yeh, and Edward "Ted" C. Lee, who encouraged and supported me through all my years in school. Special gratitude goes to the doctors and nurses at the Charleston Area Medical Center (CAMC), who helped my mom recover from her subarachnoid hemorrhage. Best wishes to Mrs. Noel and her family as well as the many other visitors and patients that I met at CAMC.

My college friends provided all the fun during my Ph.D. work and they must be mentioned. They include Tom Brooks, Christina Combs, Sara Duven, Paul Johnson, Steve Henry, Jason Hess, Rhett Hudson, David Lehn, Dan Lough, John Lund, Carl Minton, Jim Peterson, Jeff Raubitcheck, Cindy Stowell, Steve Swanchara, Dan Talabar, and Changyong Yang.

My colleagues, which have worked with me on many projects unrelated to this dissertation, must also be noted. These include but are not limited to, Phil Benchoff, Ben Cline, Pat Donohoe, Laura Durrett, Bob Fink, David Green, Carl Harris, Charles Hickman, Makaki Hirabaru, Gerald Karam, Roni Khazaka, Dorian Kim, Bob Lineberry, Pedro Roque, and Alex Swain.

Thanks to `crunch.engr.vt.edu`, `strat.visc.ece.vt.edu`, and the many other departmental machines for running many hours of simulations and dissertation writing. Without these computational resources, I would have never finished. Thanks also to Farooq Azam, Scott Harper, Randy Marchany, Mohammed Osman, and Sami Elhini for keeping departmental machines up and running.

The Bradley Department of Electrical and Computer Engineering and Dr. Nunnally provided me with the opportunity to teach courses, which was an enjoyable experience. Thanks also to the many faculty and staff members who helped out with this and other activities.

The funding provided by the National Science Foundation to SUCCEED (Cooperative Agreement No. EID-9109853) helped keep me in school for the past several years. SUCCEED is a coalition of eight schools and colleges working to enhance engineering education for the twenty-first century.

TABLE OF CONTENTS

ACKNOWLEDGMENTS III

LIST OF FIGURES IX

LIST OF TABLES X

CHAPTER 1. INTRODUCTION 1

1.1 Motivation 1

1.2 Approach and Results 2

1.3 Organization 2

CHAPTER 2. RELATED WORK 4

2.1 Active Networks 4
2.1.1 Extensible Operating Systems 4
2.1.2 Mobile Software Systems 5
2.1.3 Active Network Research 6

2.2 Interface Resolution and Retrieval 8
2.2.1 The Namespace Problem 8
2.2.2 Intelligent Agents, Web Servlets, and Agent Communication Languages 9
2.2.3 Interface Registries 9
2.2.4 Security Considerations 10
2.2.5 Network Caching 11

2.3 Summary 12

CHAPTER 3. APPROACH 13

3.1 Active Network Descriptive Model 13
3.1.1 Active Programming Languages 14
3.1.2 Active Network Operating System (ANOS) Model 15
3.1.3 ANOS Handlers 17

3.2 Resolution Approach 17
3.2.1 Network Node Resolution Architecture 18
3.2.2 Namespace Issues 19
3.2.3 Active Code Transport and Retrieval Mechanisms 20
3.2.4 Library Dependencies and Signatures 21
3.2.5 Query Language Issues 22
3.2.6 Registry Issues 23

3.3 Experimental System 25

3.4 Summary 26

CHAPTER 4. ACTIVE LIBRARY RESOLUTION SERVICE DESIGN 27

4.1 Basic Primitives 27
4.1.1 Required Active Code Primitives 27
4.1.2 Operating Environment Primitives 28
4.1.3 Active Library Resolution Service Primitives 28

4.2 Protocol Design 29
4.2.1 Active Transport Protocol 29
4.2.2 Active Library Resolution Protocol 30

4.3 Active Transport Protocol 32
4.3.1 Reliable Transport 32
4.3.2 Header Representation and Processing 32
4.3.3 Naming Rules 34

4.4 Active Library Resolution Protocol 36
4.4.1 Library Resolution 38
4.4.2 Signature Resolution 38
4.4.3 ALRP Queries 40
4.4.4 Active Registry Query Language 41

4.5 Summary 43

CHAPTER 5. EXPERIMENTAL AND SIMULATION SYSTEM 44

5.1 Experimental System 44
5.1.1 Implementation Status 44
5.1.2 Protocol Verification 44

5.2 Simulation System 48
5.2.1 Network Generation 49
5.2.2 Source Node Operation 49
5.2.3 Server Node Operation 50
5.2.4 Router Node Operation 51
5.2.5 Operation and Limitations 51

5.3 Summary 51

CHAPTER 6. RESULTS 52

6.1 ATP Experimental System Evaluation 52
6.1.1 Extensibility 52
6.1.2 Overhead and Response Time 54
6.1.3 Policy Constraints 55

6.2 ALRP Experimental Prototype Evaluation 55

6.2.1 ALRP Content Analysis 56
6.2.2 Experimental Measurements and Simulation Validation 58

6.3 ALRP Simulation Evaluation 59
6.3.1 Simulation Setup 59
6.3.2 Extensibility, Constraints, and Overhead 62
6.3.3 Resolution Time and Scalability 63
6.3.4 ALRP Error Control versus Continuous Retransmission Simulation Results....................63

6.4 ALRP Continuous Retransmission Simulation Results 65
6.4.1 ALRP Performance Analysis 65
6.4.2 ALRP Packet Loss Performance Analysis 67
6.4.3 ALRP Scalability Analysis....................68

6.5 Summary....................70

CHAPTER 7. CONCLUSIONS 71

7.1 Objectives 71

7.2 Approach 71

7.3 Results....................72

7.4 Contributions 72

7.5 Future Research....................73

REFERENCES.................... 75

APPENDIX A. OPERATING ENVIRONMENT PRIMITIVES 84

A.1 Environmental Services....................84
A.1.1 Kernel Services 84
A.1.2 Resource Services 86

A.2 Network Services....................86
A.2.1 Device Interface Primitives 86
A.2.2 Socket Primitives....................89

A.3 Resolution Handler Primitives....................89

APPENDIX B. ACTIVE TRANSPORT PROTOCOL.................... 94

B.1 Overview 94
B.1.1 Requirements....................94
B.1.2 Protocol Transactions....................94
B.1.3 Acknowledgment/Negative Acknowledgment....................94
B.1.4 Ping 95
B.1.5 Get....................95

B.1.6 Put/Post 96
B.1.7 Remove....................96
B.1.8 Response....................97
B.1.9 Status 97
B.1.10 Abort 97
B.1.11 Redirect 97
B.1.12 Dictionary....................97
B.1.13 User-Defined 97
B.1.14 Compatibility....................98

B.2 ATP Over UDP....................98
B.2.1 ATP Packet Format 98
B.2.2 State Transition Diagrams 100
B.2.3 Transaction Processing....................101

B.3 ATP Over TCP 102
B.3.1 Message Format 102
B.3.2 Transmitter Algorithms 102
B.3.3 Receiver Algorithms....................103

B.4 Header Rules 104
B.4.1 Augmented BNF Rule Set 104
B.4.2 Basic Rules....................105
B.4.3 Extension Rules....................106
B.4.4 Basic Date and Time Rules 106
B.4.5 Basic Domain-Set Rule 106
B.4.6 Active-Version Rule 106
B.4.7 Internet Media Types....................107
B.4.8 Accept Rule 107
B.4.9 Content Rules 107
B.4.10 Naming Rules 108
B.4.11 Platform and Revision Rules 109
B.4.12 Miscellaneous Information Rules 109
B.4.13 Signature Rule 110
B.4.14 Interface Rules....................110
B.4.15 Encryption Rules 112
B.4.16 Payment Rules 112
B.4.17 Authentication Rules 113
B.4.18 Restriction Rule....................113
B.4.19 Pragmas 114

B.5 Translation and Proxy-Servers....................114

APPENDIX C. ACTIVE LIBRARY RESOLUTION PROTOCOL.................... 115

C.1 ALRP Operation 115
C.1.1 ALRP Packet Types 115
C.1.2 MTU issues 117
C.1.3 Proxy-Servers 117

C.2 Algorithms 118
C.2.1 Client Algorithms 118
C.2.2 Server Algorithms....................119

C.3 Query Mechanisms and Language **119**
C.3.1 Pragmas 120
C.3.2 Constraint Categories 120
C.3.3 Dictionary Compression 121

APPENDIX D. EXPERIMENTAL AND SIMULATION DATA **122**

VITA **126**

LIST OF FIGURES

FIGURE 1. OPERATIONAL MODEL FOR AN ACTIVE NETWORK. 14
FIGURE 2. BLOCK DIAGRAM OF THE IDEAL OPERATING SYSTEM AT A NODE. 15
FIGURE 3. BLOCK DIAGRAM OF THE RESOLVER HANDLER. 18
FIGURE 4. ALRP OPERATIONAL MODEL. 36
FIGURE 5. ALRP OPERATIONAL MODEL WITH SERVER 1 AS A PROXY. 37
FIGURE 6. EXAMPLE QUERY REQUEST FOR AN ACTIVE LIBRARY. 42
FIGURE 7. CONFIGURATION FOR VERIFICATION TEST. 45
FIGURE 8. HEADERS FOR THE PING APPLICATION. 45
FIGURE 9. PROXY-SERVER (GUITAR) TRACE FOR VERIFICATION TEST. 46
FIGURE 10. CLIENT (OCARINA) TRACE FOR VERIFICATION TEST. 47
FIGURE 11. SERVER (STRAT) TRACE FOR VERIFICATION TEST. 48
FIGURE 12. EXAMPLE HIERARCHICAL NETWORK. 49
FIGURE 13. SIMULATION (S) AND PROTOTYPE (P) RESOLUTION TIME COMPARISONS. 59
FIGURE 14. ALRP RESOLUTION TIME FOR ERROR CONTROL (EC) VERSUS CONTINUOUS RETRANSMISSION (C) SCHEMES. 64
FIGURE 15. ALRP PACKET COUNTS FOR ERROR CONTROL (EC) VERSUS CONTINUOUS RETRANSMISSION (C) SCHEMES. 64
FIGURE 16. ALRP RESOLUTION TIME ANALYSIS FOR ALL NETWORKS AND ERROR RATES. 66
FIGURE 17. ALRP RESOLUTION TIME ANALYSIS FOR ALL NETWORKS AND ERROR RATES. 66
FIGURE 18. ALRP LOSS PERFORMANCE ANALYSIS FOR RESOLUTION TIME. 67
FIGURE 19. ALRP LOSS PERFORMANCE ANALYSIS FOR NUMBER OF TRANSMITTED PACKETS. 67
FIGURE 20. ALRP RESOLUTION TIME SCALABILITY. 68
FIGURE 21. ALRP PACKET COUNT SCALABILITY. 69
FIGURE 22. DEFINITION OF ACTIVE RECORD STRUCTURE. 93
FIGURE 23. ATP PACKET FORMAT. 99
FIGURE 24. SENDER STATE TRANSITION DIAGRAM. 100
FIGURE 25. RECEIVER STATE TRANSITION DIAGRAM. 101
FIGURE 26. ALRP BASIC HEADER AND ANNOUNCE HEADER FORMAT. 116
FIGURE 27. ALRP REQUEST HEADER FORMAT. 116
FIGURE 28. ALRP REDIRECT HEADER FORMAT. 117

LIST OF TABLES

Table 1. ATP Payload Types/Action Types 31
Table 2. ATP Dictionary Translation Table 33
Table 3. Query Categories 43
Table 4. ATP and HTTP Comparison 54
Table 5. Header File Analysis Results 56
Table 6. Library Symbol Analysis Results 57
Table 7. Test Network Characteristics 61
Table 8. Test Network Error Characteristics 61
Table 9. Loss Rates For Error Analysis 62
Table 10. Resolution Handler Control Options 90
Table 11. ATP Protocol Transactions 95
Table 12. Response Status Codes 96
Table 13. Query Categories 121
Table 14. Header File Analysis Source Data 123
Table 15. Error Control Versus Continuous Analysis Source Resolution Time Data 124
Table 16. Error Control Versus Continuous Analysis Source Packet Count Data 124
Table 17. Source Resolution Time Data 124
Table 18. Source Packet Count Data 124
Table 19. Loss Analysis Source Resolution Time Data 125
Table 20. Loss Analysis Source Packet Count Data 125

Chapter 1. Introduction

A fundamental feature of the Defense Advanced Research Projects Agency's (DARPA) active network initiative is that active networks will allow for the rapid injection of new protocols into an existing network [1]. An active network is extensible in the sense that switching nodes can be dynamically programmed, through the use of a platform-independent software system, to perform "custom computations" on network data. The end result is that nodes within the network can have new protocols installed "on-the-fly." This reduces protocol deployment time and can increase network efficiency. Current active network systems [2-7] work on the assumption that code is somehow delivered to the node being programmed. Other than a remark about a simple code resolution strategy in [2, 8-10], current research ignores the fact that the code may already be present, in another form, at the node or that the code may require other libraries that are not present. This research seeks to solve the latter problem by providing a global Internet library resolution and retrieval service.

1.1 Motivation

Active networks use platform-independent mobile programming techniques. Mobile programming techniques allow the reuse of the same program encoding on different systems. This program code, or active code, must be delivered across the network using some transport mechanism and automatically inserted into the receiving network node. For example, if a user wishes to use protocol XYZ, then the code for protocol XYZ is sent along with the data, installed upon receipt, and executed. However, it is probable that this will not be the most common operating mode. The first problem is that if two users want to use the same protocol, duplicate code is sent. The second problem is that if the "XYZ" protocol depends on another protocol, "ABC," then the "ABC" protocol needs to be sent as well. Both problems result in a significant amount of overhead. A better operating mode would be to send a small program fragment that, if necessary, dynamically bootstraps the required protocols. A slightly more efficient method would be to send only the protocol identification information and have the node resolve and then retrieve the necessary protocols. The questions that are evident in both of the last two operating modes are how a node would resolve the library interface and from where a node would retrieve the active code. This research addresses these two issues.

While the active network model used for this research makes several unique contributions of its own, it is not the focus of the research. The observation that efficient active networks will most likely need dynamically obtainable code from a global Internet is the foundation for the research. The active library resolution service is a distributed database problem with the data being protocol program code and the search features being a resolution of programming interfaces and object symbol tables for active networking. While not directly investigated by this work, the resolution service could be applied to standard application programming, distributed programming, mobile software agents, World-Wide Web applications, and extensible operating system libraries.

1.2 Approach and Results

The major focus of this work is the resolution strategy for active libraries, which encompasses the network query for the library to be found, the semantics of the query itself, the definition of the function primitives, and the resultant transfer of the library. A critical aspect is to ensure that the network query is scalable over an internet of a few nodes to thousands of nodes. The use of caching mechanisms in this work is discussed but not evaluated. The design of this system, which is proposed for use in active networks, is documented in Chapter 4.

Active networks will generally rely on a highly flexible form of an operating system, as discussed in Chapter 2. Potential strategies and features of an active network operating system needed for an active library resolution service are detailed in Chapter 3. To implement a resolution service, some mechanism to transport the code must be available. While not the primary focus of this research, transport mechanisms for active code were studied and a possible strategy was developed and evaluated. This strategy is based on using descriptive headers and can be applied to many different situations.

An experimental prototype was implemented that supports important functional aspects of the proposed system, including the automatic resolution and execution of required code and the proxy-server functions. A simulation model was also implemented to evaluate the performance of the service on various sizes of hierarchical internets. Both of these systems are described in Chapter 5. The prototype was used to show that the concepts can work and to provide some validation of the simulation system. The criteria of extensibility and support of policy constraints and the metrics of overhead, resolution time, and scalability were applied to the active code transport protocol and active library resolution protocol. For both protocols, extensibility and policy constraint support are designed into the system and are discussed in Chapters 4 through 6. The remaining metrics and results are discussed in Chapter 6. The resolution time and overhead for the implemented transport protocol are comparable to the widely used HyperText Transfer Protocol (HTTP) [11]. Scalability is not applicable to the transport protocol. For the resolution protocol, simulation studies indicate that the protocol is scalable for both increasing network size and error rates. For a unit increase in the distance between the requesting client and the nearest server, the resolution time increases by roughly a unit factor for the cases studied. By design, the resolution protocol has low overhead.

1.3 Organization

The remainder of this dissertation is organized as follows.

- Chapter 2 presents an overview of prior and current research related to an active library resolution system. Research on active networks, extensible operating systems, and mobile software systems is reviewed. Remote method and object resolution techniques are discussed as a related research topic.
- Chapter 3 discusses how the entire system could be built. The framework for the proposed experimental active network is presented, along with the model for the distributed resolution service. Research goals and measurement methods are also discussed.

- Chapter 4 reviews the proposed design. The active transport protocol and the active library resolution protocol are discussed. Appendices B and C provide specifications for the protocols.
- Chapter 5 presents the experimental prototype, the simulation system, and the evaluation methodology. Features that are implemented in the prototype and how the prototype was verified are discussed. The algorithms and operational limitations of the simulation system are presented.
- Chapter 6 presents the results of evaluation of the active transport protocol and the active library resolution protocol. Experimental measurements from the prototype system are used to provide some validation of the simulation system. The data content for the resolution protocol was analyzed and is used to validate fundamental design assumptions.
- Chapter 7 summarizes the work, discusses contributions of the research, and presents potential future work.

Chapter 2. Related Work

This chapter discusses relevant research related to active library resolution for active networks. Current research in active networks is reviewed along with related concepts from work on extensible operating systems and mobile software systems. Ideas from distributed computing and intelligent agents, as they relate to the library resolution problem, are also discussed.

2.1 Active Networks

An active network can be viewed as a large homogeneous programmable computation device. Each node in the network can be considered to be a processor in a massively parallel computer. The active network program code in this particular computer has two properties. The first is that it can be dynamically loaded. The second is that the same code can be used regardless of the underlying processor. Tennenhouse, *et al.*, motivate the need for active networks in [1]. Modern applications have the need for flexible switching nodes. For example, firewalls are currently specialized devices that must be custom programmed for every new network protocol. Web applets are forcing the development of platform-independent computing. These and other applications are driving the need for more flexibility in devices attached to networks.

From the literature [1-7], it is obvious that a number of related research areas exist. Each proposed or existing active network is related to or uses an extensible operating system. Another necessary element is a system in which program code is mobile. That is, the mobile program code, or active code, must be able to execute on any hardware-platform. Because mobile code is generally not efficient, a related area of work is dynamic code compilation and retargetable compilers [2, 3, 5, 7]. The effect of dynamic compilation is to convert mobile code into native machine instructions "on-the-fly." Retargetable compilation simply allows reuse of the same compiler to produce program code for different hardware platforms. Another related execution issue is code safety [2, 3, 5, 7]. Code safety addresses the need to control the execution of program code so that the integrity of a computer system is not compromised.

The remainder of this section discusses the related library resolution service issues of extensible operating systems, mobile software systems, and active networks. These systems are assumed and required services that must be present in order for a library resolution service to work. Related research for the resolution service is reviewed in Section 2.2. In the remainder of the dissertation text, the term active code is used to refer to both active programs and active libraries. The term active program is considered an executable application whereas an active library is a module, typically a network protocol, used by an active program.

2.1.1 Extensible Operating Systems

The original monolithic operating systems consisted of one single program kernel image that was hard to modify and resulted in inefficient memory and resource usage. The next operating system development was the micro-kernel. Micro-kernels reduced the amount and type of kernel

code to core features such as memory management and scheduling. The current evolution in operating systems design is to further reduce the core features to just hardware interface protection abstractions. Users customize the operating system by providing most of the various user-level micro-kernel abstractions such as interprocess communications (IPC) and network protocols. This type of operating system is generally referred to as an extensible operating system. Four active network related extensible operating systems are discussed below.

- The MIT exokernel [12-15] system separates resource management from resource protection, allowing the operating system to be customized. The kernel provides the base resource protection mechanisms while user-supplied code provides specific resource management implementations. This allows for new service interfaces, typically grouped into some operating system library, to be inserted into the operating system to achieve a better level of performance and flexibility.
- Scout [16] is another extensible operating system that focuses on improving network communications performance within an operating system. Scout seeks to reduce the computational cost for network code, in a scalable manner, through the use of more efficient abstractions and better compiler technology.
- Another extensible operating system is SPIN [17, 18], which uses dynamic linking techniques and the Modula-3 programming language to build the operating system. An extensible TCP/IP protocol implementation used with SPIN shows a significant execution performance improvement when compared to a monolithic Digital UNIX implementation [19].
- Shapiro, *et al.*, have built a test active network implementation in EROS [7]. They also provide some requirements for active network operating systems, which include real-time operating system support, high network and IPC performance, fine-granularity memory protection, and protection and code provability.

Production UNIX micro-kernels that have extension features include Solaris and Linux. These use the concept of dynamically linkable modules to achieve extendibility. These modules must be stacked in the order of dependencies. An extensively modified Linux modular extension capability could be used as a primitive active network extension mechanism. However, this is not performed in this research, which uses a simple user-space application to simulate parts of an active network operating system. Active networks do not need new operating system architectures — new architectures simply increase the execution efficiency of the system in question.

2.1.2 Mobile Software Systems

Mobile software technology forms the basis of active networking by providing platform-independence and a higher level of abstraction for network programming. There are two general categories of mobile software systems, systems where program code is interpreted and systems where program code is dynamically compiled into native machine code. Virtual machine code is a type of interpreted code and is not considered, by this research, to be a separate mobile software system category. For many interpreted systems, code can be compiled into

native machine code. This dynamic compilation concept is called just-in-time (JIT) compilation or dynamic code generation.

Before active network-based software systems are discussed, a classification for mobile software systems is reviewed. There are three criteria that can be applied to any mobile coding system [1-2]. These are mobility, safety, and efficiency. Mobility is defined as the ability to transfer and execute programs on different platforms. Safety is defined as the ability to restrict resource access. Efficiency is defined as having mobility and safety with acceptable performance.

Mobility is generally achieved by using a machine-independent program code and is an inherent property of any mobile software system. Machine-independent program code is generally slower than machine native code. Native code at the kernel level can access any memory location in the system. Native code at the user level, due to hardware and software protections, can only access memory within the user space. Regardless, any native code program may cause catastrophic system failure. Safety can be increased through the use of interpreters and other hardware and software protection schemes. The cost of increased safety is generally lower efficiency. Or, vice-versa, native code systems have the best efficiency at the cost of safety.

A number of interpreted code systems, such as Java [20], NetScript [3], and Safe-TCL [21], have been proposed for use in extensible networks. There are two types of interpreted systems, high-level program code interpreters and intermediate program code interpreters. High-level program code interpreters are those that execute based on the program code itself and do not attempt to compile the program code to some intermediate byte-code format. Two dynamic compilation systems that are being investigated in active network research are Java and C. In many respects, it could be argued that Java and C are functionally equivalent. The Sumatra project [22] is investigating a Java to C converter, which would then compile the resulting C code to natural binary code. The Liquid Software project shows that it is possible to compile simple Java code to native code, on a 200 MHz machine, in the time it takes to receive the code on a 10 Mbps network connection [5]. Many researchers [23-31] show that dynamic code generation may increase application performance or investigate dynamic compilation techniques. From these results, it should be obvious that dynamic compilation techniques can be used to achieve mobility and good performance.

Additionally, a number of safe execution, dynamic linking, and code extension strategies are available in the literature [14, 15, 32-42]. Existing safe code execution, extension, and linking strategies can, with sufficient effort, be used in developing an active network system. Thus, these are not investigated in this work.

2.1.3 Active Network Research

The previous discussion on mobile software systems and extensible operating systems are precursors to the development of active networks. Before active network research is reviewed, the two ways to view active code are discussed. As outlined by Tennenhouse, *et al*. [1], the first view is that active code is program instructions that are embedded into a packet. The second

view is to consider every packet as a program, or a capsule. The subtle difference is that, in the latter system, each packet must contain specific instructions on what to do with itself. Either method could leave behind persistent code, which would reduce the amount of extension code for future packets. There is a significant amount of overhead for raw data transfer for the capsule method; however, with increasing computational power and network bandwidth, this should not be a significant problem. The capsule approach has the benefit of providing a higher level of network abstraction and the higher level of abstraction allows increased ease-of-use [1, 6].

The MIT ActiveNet [1, 2, 43] project could be considered the leading project among all active network research. The researchers conceptualize the use of a capsule-based programmable intermediate network node, or a programmable network switch. In [2], they discuss the types of information and features, such as routing tables and persistent storage, that the capsules must be able to access. A number of security questions related to such information access are raised. The MIT approach is to design an IP-based delivery mechanism [44] and a simple Java-based implementation using Linux. They also discuss a potential cache-based demand loading scheme to resolve protocol objects. The proposed MIT approach differs from this research in that the MIT system depends on the set of previous nodes, that the active program traversed, to provide the necessary active library. This research allows arbitrary nodes in the network to provide the libraries. There are a number of problems that may result in the MIT approach. One problem is that if the previous node no longer has the required active code, then the active code cannot be executed at the current node. Assume a new application program, that uses active code modules, is obtained and installed on a system. If this application assumes the use of a protocol that is not on the current system, the MIT paradigm fails. In the new application case, the only readily defined source is the author of the application. Also, obtaining the protocol from a local site, as opposed to a remote site, can be much more efficient.

The Liquid Software [5] and EROS [7] project were described in the previous section. Conceptually, they are the same as the MIT project but the related research focuses on different issues. Liquid Software is based on the Scout operating system and focuses on compilation techniques and function primitive requirements. EROS examines performance related features and requirements for active networks. The NetScript project [3] is another active network project that focuses on the development of programming language features specifically needed for an active network. Their approach is to develop a new programming language, NetScript, to test their ideas. Researchers at the Georgia Institute of Technology are looking at how active networks can be used to reduce congestion [4]. SwitchWare [6] is focused on a network programming model for network switches. SwitchWare is based on the Bellcore Advanced Intelligent Networking approach as opposed to the Internet Protocol approach. They discuss a number of requirements for active networks and also examine security issues. Switchware researchers have implemented in ActiveBridge in which new modules can be remotely installed [45]. All module dependencies must be resolved by a remote operator.

2.2 Interface Resolution and Retrieval

The specific problem that is investigated by this work is active library resolution and retrieval. The assumption is that a given node in an active network must locate and obtain an active library. The active program makes a query to the network resolution service and the network resolution service attempts to resolve the query. If a positive result is obtained, then the node retrieves the active library. A simple resolution strategy is remarked on in [2]. This simple strategy depends on the set of previous nodes that sent the active program to provide resolution services. One problem with this model is that if the previous node goes away before the dependent code is retrieved, the active program is useless. A related problem is that if there is no source node, such as in the case of a newly loaded application, the strategy does not work. Another problem occurs when crossing network protocol boundaries. For example, if the source uses only TCP/IP and the destination uses only IPX, the code is useless even though both machines may be connected by a common TCP/IP and IPX machine. The proposed strategy resolves active code from the network; thus, a TCP/IP interface can be found for the IPX machine or an IPX interface can be found for the TCP/IP machine. Restrictions and other issues related to active library resolution are also not discussed in [2].

As noted earlier, the problem of active library resolution is essentially searching a distributed database or registry. How to search, or query, the registry is a complex task. There are numerous criteria that could be used to define the “correct” library to use and there may be an infinite number of matching libraries. The query mechanism must be general enough to handle these situations. In addition to query and retrieval, the registry must have mechanisms for users to add, modify, and delete library information. Security of the registry and whether or not to execute the obtained active code is another concern. However, the most important metric is the scalability of the system. This section reviews related research and concepts regarding these issues.

2.2.1 The Namespace Problem

Each library interface and each primitive, or function, must have some identifier, or name. Proper management of this set of names, or namespace, is critical for global scalability. For example, the GNU *libc* [46] is a publicly available C library. Machine-specific implementations aside, one can assume that GNU would provide a registry source for the library. Assume, for some reason such as improved performance or security, that other organizations may wish to mirror the library. Also assume that a site has an optimized local version of the C library. Ideally, for scalability, the location information and the library itself would be stored in a local on-site cache. If everybody used “*libc*” as the registry name then there would be no serious problem to obtain the library from the GNU registry source and mirrors. However, there is a namespace collision for the optimized local library.

In the area of distributed computing, there are a number of namespace management methods [47-51]. The general concept exhibited in each system is that names may have special meaning within one group that is different from another group. Additionally, the same name may have a different global meaning among all groups. The basic solution is to associate some context with the name that differentiates it from each group. The context essentially provides a mechanism to

separate the namespace. This is not a one-to-one mapping to library interface problem. Primitives in the library and primitives that are expected by the calling routine may not be the same.

Two types of namespace matching are used in the proposed system. The first is the absolute name of the library, such as "GNU *libc*," and the second is the generic name of the library, such as "*libc*." The primitive matching problem may be partially solved by matching function signatures, in which a signature consists of the function symbolic name and parameter types. There is no real guarantee that the obtained library is truly the one that will work. These and other criteria simply increase the likelihood of success.

2.2.2 Intelligent Agents, Web Servlets, and Agent Communication Languages

Intelligent agents can be defined as autonomous high-level software systems that perform specific activities for a user or another agent. Intelligent agents may greatly benefit from the creation of an active network. They may also benefit from the proposed resolution service which could be extended to support intelligent agent retrieval. Certain aspects of mobile agent research, which is a sub-branch of intelligent agent research, has similarities to active network research. Both use mobile programming systems and a network — they just operate at different levels. Stated differently, mobile agents are autonomous applications that could use the active network infrastructure. For example, IBM's Java-based Aglets [52] allow transport and execution of program code across multiple machines. The Aglet interfaces do not provide low-level access to hardware and operating system code. More information about intelligent agents can be found in [53-55]. Discussion of the usefulness and requirements of mobile agents can be found in [56, 57]. A related area of work in the Web is the use of "servlets" [58]. Servlets are mobile programs that are transferred from Web server to Web server and perform some function for the host server. Similar to mobile agents, servlets effectively operate at a higher protocol layer.

Because inter-agent communication is an important issue for intelligent agents, much work has been performed in the area of agent communication languages. Issues in agent communication languages can be found in [59, 60]. A number of agent communication languages are in existence [61-66]; however, they are unsuited for the specific task at hand. They are either too general purpose or are too domain specific. This research uses a simple and extendible query language format.

2.2.3 Interface Registries

Distributed systems are essentially a collection of networked systems in which each part of the system helps to complete some part of an overall task. There is one aspect of distributed systems that is of particular interest in this research, how to call remote procedures and methods. The data transfer and conversion and other specific programming issues related to remote calls are not of concern for the proposed resolution system.

The most popular remote call system is Sun's Remote Procedure Call (RPC) [67, 68]. RPC provides a simple procedure resolution service in RPCBIND [69]. DCOM [70], CORBA [71], and the Java Remote Method Invocation (RMI) [72] also have remote call capabilities and provide more complex naming and registry services. The above general remote call and registry functions are loosely based on the Distributed Computing Environment (DCE) model [48, 73].

Each of the four systems has differing approaches to namespace division. CORBA divides its namespace into multiple domains and each domain has unique identifiers; however, the identifiers do not need to be unique across domains. One can have a registry for each domain. In DCE, the domains are called cells. The effect is similar to the name-context division described in the previous section. RPC and DCOM use direct queries to the host in question. The host must be known in advance and the namespace is effectively divided on a host-by-host basis as opposed to a global namespace among all hosts. Java RMI uses Uniform Resource Locators [74] to specify the library object in question. Again, the host must be known in advance and the namespace is host dependent. One reason why this is not an ideal model is that if a network-local copy of the library is available, there is no mechanism for the application to get the network-local copy. Another reason is that the service fails if the library is no longer available at the source even if the library exists at other network nodes. CORBA provides the most powerful namespace and registry search model, which divides the namespace into a set of domains. These domains, or contexts, are application specific and applications can use multiple domains. Global search capabilities are not generally supported by the three systems; however, it would be straightforward to add that capability. The approach used in this research is most similar to CORBA in that it divides the namespace into a name-context pairing.

Caching of registry information and library objects help improve scalability and does not exist on the above systems. Caching reduces the load on the network and avoids the creation of a small number of globally critical systems that quickly become bottlenecks. This problem could also be solved in each system. Caching is discussed in Section 2.2.5.

A related area of development is the Internet Service Location Protocol [75]. This service supports the naming and location of network services through the use of the Domain Name System (DNS) [76-78]. The result is fundamentally the same as the Java RMI usage of Uniform Resource Locators.

2.2.4 Security Considerations

Three common security features are discussed below. The features are authentication, integrity, and confidentiality, and are required features in the proposed system. These features are properties that may or may not be present in any given cryptographic algorithm.

- *Authentication* is the property of being able to verify that the originator of the active code is indeed the real originator.
- *Integrity* is the property of being able to verify that the active code is the unmodified and complete original copy.

- *Confidentiality* is the property of being able to conceal the active code from a third party that intercepts the transmission.

Authentication and integrity are the two features that the active code delivery and retrieval model must have. For example, this allows the user to verify that the GNU *libc* library that they just obtained did indeed come from GNU and came without any modifications. Normally, it is not important to have confidentiality. However, considering commercial aspects of library code, some payment scheme for library use is necessary and that transaction must be protected. It cannot be assumed that the transport protocol provides security services, such as those provided for in the IP security architecture [79]. This is because most internet protocols do not currently have or require such support.

2.2.5 Network Caching

Caching is one of the most frequently used mechanisms to obtain performance scalability for network services. A large number of systems have some form of caching, from incremental updates to complex prediction and replacement algorithms. Incremental updates have mostly been used in network layer protocols such as routing [80, 81], workgroup management (NIS) [82], and name service [76-78, 83]. One of the more complex caching protocols is the one used by the Web in HTTP version 1.1 [11, 84].

In general, there are three types of caching. These are post-access, replication, and predictive caching. Post-access caching occurs when a user requests some item of data from a server and the server stores a copy of the data for future requests. Replication caching is making a complete copy of the database. Predictive caching means that data in the cache has not yet been requested and the server obtains a copy based on the likelihood of the user requesting the data in the future. The cache system in the proposed system allows the use of all three caching mechanisms.

There are a number of considerations when designing a caching system, including cache coherency, cache replacement policies, and commercial features. A brief overview of these issues is presented here with related discussion in Chapter 3. The overall thrust of cache coherency is to ensure that the data in the cache is an exact and current copy of the real data. In the application under study, the likelihood of data removal or data addition is significantly higher than data modification. This is due to the nature of protocol libraries — changes take form as new releases, which generally change version numbers. Thus, keeping the cache coherent at a reasonable expense is relatively straightforward, unlike an application with highly variable data such as HTTP. Since the cache servers have finite resources, an important question is when to expire data and replace the data with something else. Expiration and replacement policies are implementation issues and should not be mandated by a specification. To assist in potential pre-fetch caching, a result from Web caching research will also be used here [85]. This result is to have a mechanism for sending cache probability information across library servers. Other issues include copyright and payment for the use of protocol libraries. These commercial aspects are more closely related to the query language design and are discussed in Chapter 3.

2.3 Summary

This chapter reviewed the general operating principles and concepts behind active networks, extensible operating systems, and mobile programming systems. Active networks, combining work primarily in extensible operating systems and mobile programming systems, have the potential to revolutionize the way the network is viewed. To ease the development and deployment of active networks, an active library resolution model needs to be developed. Related concepts that are used in the development of this model were also presented.

Chapter 3. Approach

As noted earlier, the active library resolution service is essentially a distributed database problem. This database, or registry, must be scalable and should be designed to be extensible. To demonstrate the operation of the system, a prototype resolution system was built. This prototype transports simple programs and allows library searches to be performed. This chapter discusses the operating environment, approaches to perform the resolution strategy, and issues related to active library resolution. The term operating environment is used to refer to an emulated active network operating system or a native active network operating system. The requirements and evaluation approach for an active library resolution service are also presented.

3.1 Active Network Descriptive Model

As discussed in Chapter 2, there are a number of requirements for active networks. However, from the standpoint of designing and evaluating a system for active code resolution and retrieval, a number of simplifications or abstractions can be used for the active network. Figure 1 shows one potential operational model of an active network and its resolution service. Another similar operational model is presented in Chapter 4. Assume the source wishes to send active code to the destination and that the active program code requires protocol XYZ. Also assume that only the source and the remote library server have the code for protocol XYZ. The network transmission path consists of one active switch. Upon receipt and examination of the active program, as shown in step 1, the active switch determines that it does not have the code for protocol XYZ. In step 2, the active switch asks the previous node for the protocol and retrieves it. Then, the switch executes the program code, which sends the program code to the destination, as in step 3. Assume that the switch runs out of buffer space and flushes the active program and protocol XYZ. The destination node, in step 4, asks the active switch for the XYZ protocol and receives a failure in return. The destination node now has a choice, whether to ask the local library server or other nodes back along the path to the source for the code. Considering that the active program may no longer be present at the intermediate nodes, the destination will ask the local library server. The server does not have the XYZ protocol but is aware of its existence. The local server returns the remote library server as the source for the XYZ protocol, as shown in step 5. The destination then obtains the XYZ protocol from the remote library server, as shown in step 6.

The outcome of step 5, that the active library may not be present at prior intermediate nodes, is a critical step. It is also a reasonable assumption that an intermediate node may no longer have the library, if it had it at all, or that the intermediate node may be down. For example, if there were ten intermediate nodes that did not have the library, then the resolution time is just increased by a factor of ten. This factor does not consider the observation that the network latency will increase relative to the distance of the intermediate node from the node making the query. Thus, on average, a library retrieval failure at the first intermediate node should result in a query to the local library server. In fact, it may be that the first and only query should go to the local library server.

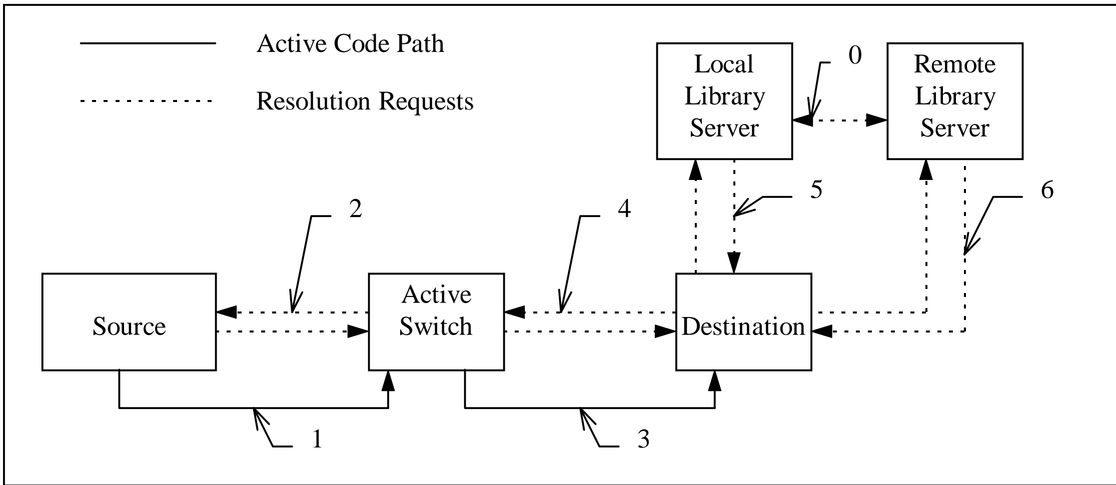

Figure 1. Operational model for an active network.

The above scenario illustrates the components required for an active library resolution scheme. Steps 1 and 3 show the need for some type of active transport protocol to get the active code from a source to a destination. Steps 2, 4, and 5 show the need for a resolution protocol and a query language to help find the correct active library. Step 6 indicates that either the active transport protocol, query system, or another mechanism must be available to retrieve the requested library. Step 0, which was not discussed in the example, is a critical background step. This step is the mechanism that servers use to trade library location and search information. Step 0 can also be viewed as how the search propagates through the network. Two obvious incidental requirements are an operating environment for the active code and a choice of a mobile programming language for the active code itself.

The remainder of this chapter discusses the required components and the evaluation of this approach to library resolution. Supporting components, which are the active programming language and the operating environment, are presented first, then the required systems for protocol resolution are discussed. The specific systems include the mechanism to transport active code and a resolution protocol that includes a query language.

3.1.1 Active Programming Languages

There are a number of programming languages available for use in an active network and, for the foreseeable future, there always will be. Thus, it is shortsighted to develop a production active network on the basis of a single language. Programming language constructs may use various levels of abstractions, but the network layer interfaces and operating system abstractions must and will be similar across all, or at least most, programming languages. For a well-designed system, compilers can translate any given programming language into an intermediate form that can be used by the hardware. The design and language requirements of active networks are being researched by other groups [3, 6, 7]. The essential requirements are access and control of the network hardware and other data structures used by the operating system's network processes.

The precise protection and access mechanisms are features of both the active network operating environment and programming language.

As presented in Chapter 2, there is a significant amount of work and progress in mobile software systems and dynamic compilation. Because of this other research [14, 15, 22-42], it is not a major concern of this work what language is eventually chosen. The end result is that the code is interpreted or compiled into a binary program specific to the local hardware. This research uses dynamically compiled C binaries, without loss of generality that some other active programming language could be used.

3.1.2 Active Network Operating System (ANOS) Model

As this research investigates one system needed in an active network, the complete implementation of a full active network and operating system is not necessary to prove that the resolution service is a valid and good approach. [1-19] demonstrate the feasibility of active networks, which provide the basis for the theoretical operating environment of the resolution service. This operating environment is called the active network operating system (ANOS). The block diagram for the assumed functionality of the ANOS is given in Figure 2. Unlike traditional operating systems and in the vein of extensible operating systems, there are three privilege levels. The user space is the same as the user space in traditional operating systems. The core kernel has the bare minimal protection functions, as in an extensible operating system. The kernel code operates at the most secure privilege level. The final privilege level is the handler space, which is between the user space and the kernel space. This is where active code would typically execute.

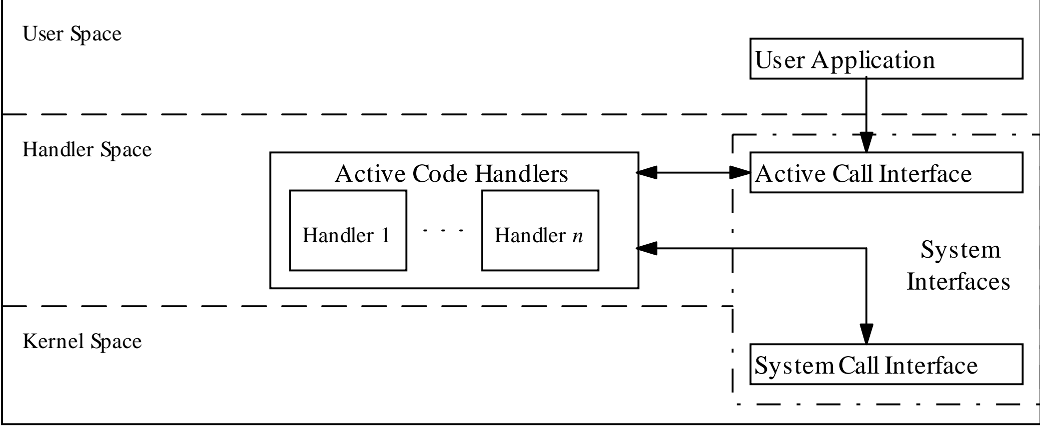

Figure 2. Block diagram of the ideal operating system at a node.

There are two types of active code considered in this research. One is considered an operating system handler, which runs in the handler space. The other is considered active user code, which runs in an operating environment similar to the environment provided by the ANOS to its active handlers. Active user code is effectively mobile agent code. The active handler is the only real active code mechanism considered in this model as it is the most efficient model. The second

active code method can be easily added to an ANOS. In some respects, interpreted code can be viewed as mobile agent code.

This research defines a handler as a module that implements some type of resource management function for an operating system, or as some transient or persistent active network code. The handlers are essentially operating system modules. However, unlike module-based operating systems, there is no ordering of handlers necessary. The handler loader, which is part of the system interface, resolves symbols before loading and inserts access protection, or guard, code on untrusted modules. If symbols are unresolved, then the resolver handler is used to retrieve the necessary code. Devices operate in the traditional manner and can be loaded as active handlers as well. One interesting idea is that the compiler can insert code that attempts to resolve references at run-time, without the need for the handler loader to attempt to resolve references. This is a possible mode of operation that is not implemented here, but could be researched by others.

The privileged interfaces, or the active and system call interfaces, must contain the necessary primitives to query, register, load, and remove an interface. Additionally, the privileged interfaces must provide primitive access for higher layers to use. The result is that system calls must be standardized so that the active code can roam the network.

To directly support multiple instances of similar, but not identical, interfaces, every active code handler must be assigned a local unique handle. An example of this is two protocols that both call themselves IP. Both the handle and the specific primitive name are used to resolve symbol references and make primitive calls. Thus, one application can be using GNU *libc* whereas another could be using Solaris *libc*. This handle concept is most similar to Sun's dynamic loading interface [86].

A remark on the difference between dynamic loading and dynamic linking may provide some additional clarification. Dynamic linking is the resolution of unresolved symbols done before execution, but after compilation and linking. Dynamic loading is the resolution of unresolved symbols performed at run-time. The strategy planned for use in ANOS is a combination of dynamic loading and linking. Some modules are dynamically linked in the sense that dependencies are resolved before execution. Others are dynamically loaded in the sense that they can specify a library to load at some arbitrary point during execution.

A simple operational procedure for loading and executing active code is presented below. This algorithm assumes that security services, compilers, and other necessary handlers are loaded.

1. Active code, or an active handler, is received from the network.
2. The ANOS performs security and access checks.
3. Assuming the checks succeed, the ANOS inspects the code for problems. If the code has unresolved symbols, then it is missing another library. Otherwise, ANOS proceeds to step 5.
4. The ANOS calls the resolver handler, which attempts to find the needed library and return to step 2 when the code is received.

5. The active code is either directly installed into the operating system, dynamically compiled and installed, or run in an interpreter.

Obviously, this high-level and theoretical overview of the operating system. Specific details, such as how to control access to physical devices, are not given or designed since this research is not attempting to design an extensible operating system. With some effort, it is possible to map this model into an existing extensible operating system such as MIT's Aegis [12].

3.1.3 ANOS Handlers

The base operating system distribution must provide a minimum of two required core active handlers in order to be able to do real work. The first required handler is straightforward — some internet protocol must be present. The second required handler is the library resolver, which uses the network protocol handler. Note that it may be possible to construct either of the two handlers as a set of smaller handlers. There are also a number of recommended handlers that the operating system should provide. User applications do not have to use any of these handlers provided by the distribution.

This base handler model differs from other active networks or extensible operating systems in that the compiler, or interpreter, is not considered an integral part of the base system. The view is that the compiler is simply another module that is "plugged-in" as desired. In fact, the view is extreme in the sense that only two handlers are required — a default network protocol in which an active transport is defined and the active library resolver. Every other handler in the operating system can be obtained from the network. Due to the nature of active code, the same operating system components can be used in any microprocessor. Additionally, if the network protocol can automatically configure itself, e.g., as the stateless automatic configuration in IPv6 [87], then the system bootstrap can be completely automatic from the time it is installed and plugged into the network. This feature is not investigated in this research as it is an incidental and beneficial result of the design approach — it has nothing to do with the active resolution approach.

Considering that users may not feel secure with the concept that their operating system is mostly loaded and tailored from libraries obtained from the network, vendors would probably distribute specific dynamic compilers, interpreters, file systems, memory mangers, devices, and so forth. Some standardization of the interfaces to these libraries must be performed; however, that is not the focus of this work.

3.2 Resolution Approach

Ideally, every active node within the network should provide resolution services. However, the level of information and length of time the information is kept varies from node to node. Some nodes will be considered active library servers that will have more persistent and detailed information whereas others will just contact these server nodes and use their services. Ideally, all nodes in the network should also be able to automatically locate nodes that consider themselves to be servers. The traditional mechanism to do this is to use some form of multicasting. Nodes

should also be able to inform requesters of other servers. The remainder of this section discusses the overall resolution strategy, the namespace division problem, and then presents major functional components such as the mechanism to transport and retrieve active code, the resolution protocol's query language, and the scheme to create a distributed registry.

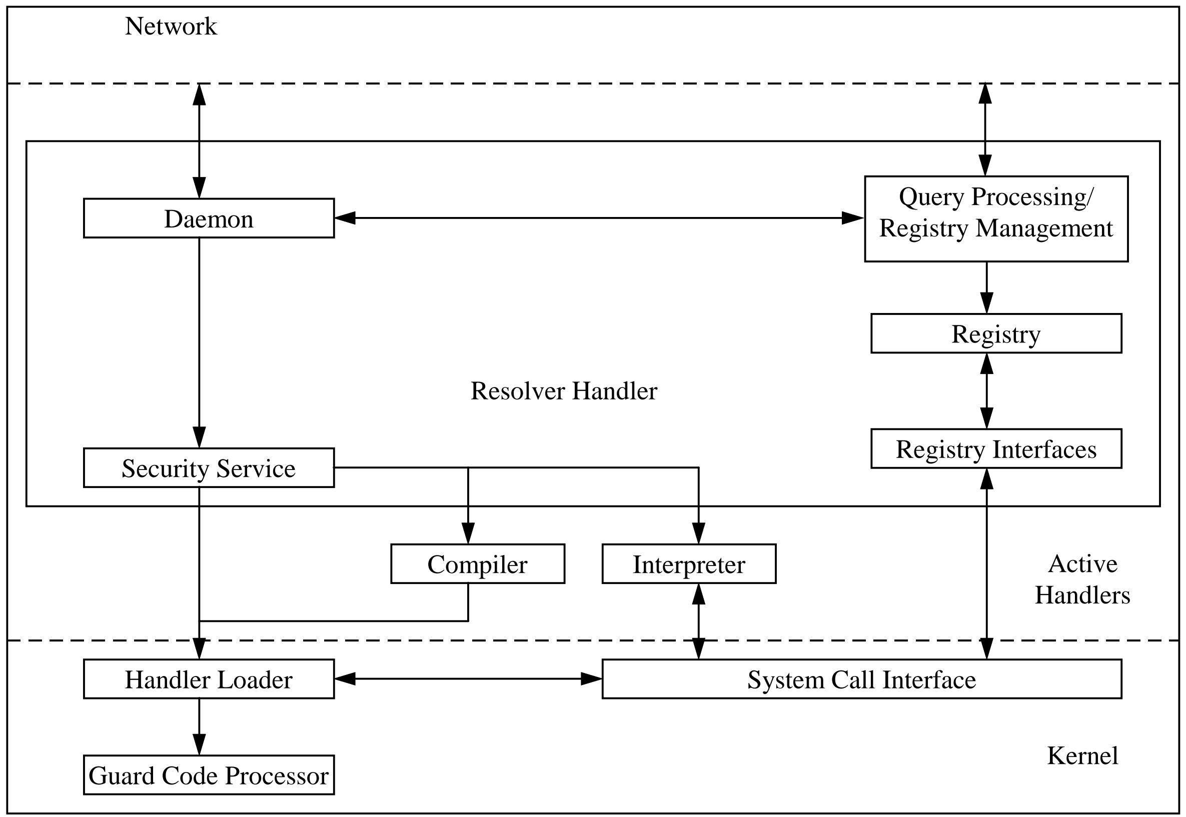

Figure 3. Block diagram of the resolver handler.

3.2.1 Network Node Resolution Architecture

Figure 3 shows the block layout for the resolver. The daemon simply waits for network service requests. If a query request is received, then the query sub-system will handle it. Otherwise, if active code is received, it is processed by the security service before it is prepared for execution. The security service may authenticate the source and check the integrity of the code. It may also check other administrative constraints. Once the code is verified, it can either be directly inserted into the operating environment, compiled and inserted, or run in an interpreted environment.

The query sub-system performs and handles both local and remote library resolution requests. Thus, the resolver handler must maintain or have access to a cache registry of the active code information obtained from the network. If it is not in the cache, a request must then be sent to the network. On the other hand, the resolver in a server mode would simply have a registry, cached or otherwise, of libraries it has on hand or that it knows about. The fact that it knows

about other servers means that some mechanism must exist to register servers and transfer data between servers. This mechanism is discussed later.

For the resolver handler to work with the operating environment, some primitive registration system must be present. This registration procedure must exist for both the operating environment and the resolver handler. This is because the operating environment must know which resolver to call for any given case and the resolver must know which network protocol to call for any given case. This strategy allows for the dynamic replacement of both the resolver handler and the network handlers. Additionally, the resolver must be able to hand over certain requests to other active code. This allows active programs to implement their own resolution strategies using the basic query language and services as defined by this research. The flexibility of the system is also evident in the numerous possible implementation methods. For example, a resolution server can choose to obtain a library and return itself as the source or return other servers as the source. The latter mechanism keeps server load down whereas the second mechanism keeps client load down. In general and considering that there are multiple views on the operation of active networks, designers can use various components of this resolution service to fit their needs. As this handler swapping ability is a function of the operating environment, it is not implemented in the experimental system.

3.2.2 Namespace Issues

As discussed in Chapter 2, names associate a library with some identifier. Proper management and definition of this set of names, or namespace, must occur in a distributed system. An example of a naming approach that could be used in the resolution service is given. Assume that there is a standardized library named *libip.active*. This library implements the Internet Protocol version 4 (IPv4) and suppose a set of standard interfaces has been defined. Because it is a registered standard name, *libip.active* has global scope. Also, suppose that different vendors have implementations that conform to the standard and one, GNU, is better than the rest. To access the GNU version, a request can be made for *libip.active* with the context GNU. If, for example, the new Internet Protocol version 6 (IPv6) has a similar library, it can be called *libip6.active*. Ideally, the old *libip.active* would also be referred to as *libip4.active*. And, as IPv6 dominates over IPv4, *libip.active* should eventually refer to *libip6.active*.

Thus, it is evident that several requirements exist. The name may or may not be unique across the internet. In the proposed resolution approach, there are three names associated with each library. The first is a general name suitable for use in a federation of computers. The second and third names are absolute and globally unique names. The second name designates the active code as provided by its source and the third name designates the active code as provided by the current node. A federation can be considered a loosely organized set of computers, each under different administrative domains.

The reason why each library must have a general name and a more specific name is illustrated by the following example. Assume that user A and user B both have an IPv4 library that has interface primitives that meet some internationally adopted standard. User C wishes to use any

library that supports the standard and user D wishes to use non-standard features available in user A's library. One identifier is not sufficient to handle this case. Further, there must be two specific names. If the library provided by user A was copied to user D, user C must be able to distinguish between the copy and the original library.

The general name has a number of properties and uses. There are two types of general names, registered and unregistered. Registered names are official identifiers that have been standardized by some registration body whereas unregistered names are identifiers that developers create and use. The name can have a generic global meaning or a specific context that may be global or may be local. The meaning of a name can change over time, as well. Generic names can refer to a library or another generic name. Absolute names uniquely identify one specific library and source on the network. Multiple different copies of the library each have a unique absolute name. Because of standardization, the ideal method to specify absolute names is to use Uniform Resource Locators [74].

It is important to note that the name of a library is not the only distinguishing characteristic of the library. There are other descriptors, such as version information, that may be used to describe various details about a library. The effective result of using the descriptors is that they provide additional context operations on the name of the library.

3.2.3 Active Code Transport and Retrieval Mechanisms

The scenario in Section 3.1 and the overall approach discussed in Section 3.2.1, raise questions of how the transport of active code is accomplished. Numerous transport mechanisms exist which include the creation of a new internet protocol specifically for active networks, to embedding active code processing as a new protocol layer in existing networks, to using existing network transports. The first approach would require extensive standardization and years of deployment. The second approach has an easier buy-in and, in many internet protocols, can be performed transparently so that legacy systems will still work. The third approach is more in the domain of mobile agents but is the simplest way to provide active code transport.

In the active network model presented above, the ideal transport mechanism is the second approach. The two resulting issues are how to integrate the active network support into the protocol and how to make the support backwards compatible with nodes that are unaware of the active network. In the Internet Protocol (IP), this is relatively straightforward to do, as demonstrated by the MIT ActiveNet project [10, 44]. Their approach is simply to use a special IP header that includes a single identifier that defines what type of programming language is carried in the packet. This research takes a different view in that more information about the active code may be needed. Note that the MIT approach does not preclude this. A programming language identifier is not sufficient since, for example, it does not provide any information as to the security of the active code. Other items that should be available, if needed, include distribution restrictions on the code, code revision information, and detailed object type information. The object type identification would identify the encoding of the active code. The encoding could be a programming language or a hardware-dependent instruction set.

Transmission of dependency information can also significantly ease resolution problems. Obviously, not all requirements for active code headers are known at this time. Some of the security issues have been recently addressed by the proposed Active Network Encapsulation Protocol (ANEP) [88]. Thus, extensibility of the system becomes a primary design objective. These headers are based on the Multipurpose Internet Mail Extensions (MIME) [89] and RFC 822 [90]. This has certain ramifications for use with HTTP and Internet mail, which is discussed later.

Active code retrieval is simply the inverse of active code delivery. The above transport context describes active code delivery but could easily describe active code retrieval. The requirements are almost the same — the only difference is that the mechanism for active code retrieval must include a parameter to specify the active code to retrieve. Since it would be inefficient to design a different protocol just for retrieval, a simple addition is made to the above transport protocol.

For a proof-of-concept prototype, it is not necessary to implement modifications to the Internet Protocol. Transport of active code is merely copying bits and each active code program can define its own bit copying operation. Given this, some generic transport mechanism that understands active networking can be used. Thus, a protocol was developed to use the Transmission Control Protocol (TCP). This protocol, with some modifications, can also be used as an IP level protocol or even as an application level protocol.

An interesting problem is what happens when an active program is waiting on a resolution and, through coincidence, the required library happens to show up. The local resolver should inform the active program of this and abort the search. This problem indicates that processing-related modules must exchange information.

3.2.4 Library Dependencies and Signatures

A fundamental feature of the proposed resolution model is the fact that it can resolve dependencies between function interfaces. However, there is one serious problem with arbitrary dependency resolution in most languages that are not object-oriented. These languages generally do not specify library information within the program code. The library information is specified during program compilation — at program link time. The danger in this is that if this information is not supplied in the active code, then all the resolver will have is a set of unresolved primitive symbols. These symbols may be from the same library or from different libraries. It is a potentially intractable problem for a resolver to resolve all the symbols to the correct libraries, given that there may be an arbitrary number of matching symbols in the libraries on the network. Thus, one important stipulation is that the essential library information must be present in the active code. This could be performed by making a registry call during active code initialization. This registry call could be embedded within a stub code generator, if one is used. However, a simpler approach is to have the compiler, or user, create the dependency information and attach it to the library information.

The inclusion of dependency information in active code also handles the multiple library retrieval case. For example, assume that the first library needs another library. Then the second library is obtained and it is determined that it needs a third library, and so forth. Dynamic loading of libraries at runtime will require either an advanced compiler or that the user make a sequence of interface dependency calls before issuing the dynamic load.

Another problem related to the active code headers is the specification of the function signatures themselves. The use of symbol names and primitive data types are fairly standard for any programming language. However, the specific primitive data types vary from language to language. One possible solution is to translate all primitive data types into an intermediate form, such as ASN.1 [91]. Another method is to use the source and language-dependent representation and translate to an intermediate form only when necessary. The only translation case is when an active program is using libraries of a different language. The dynamic translation approach is used in the proposed system. This is because active programs will probably have the tendency to use libraries developed with the same language as opposed to a different language. Thus, translation to ASN.1 for all cases is an unnecessary and expensive use of processor resources.

3.2.5 Query Language Issues

The query language is the basis of the resolution service. The language describes the library that is being sought and how to obtain that library. The query is performed using a protocol that is independent of the protocol used for active transport. The query language must query library headers that are extensible, which implies that the query language itself must be extensible. The problem is compounded since new library representations will add new search criteria. An additional problem is that users may want to impose constraints onto the query. For example, a user may not want to get a library for which she or he has to pay. Thus, administrative and commercial policy constraints must also be supported by the language. There are as many possible constraints as there are people; possibly more constraints than people. Thus, it may not be possible to design a language that is suitable for stipulating all constraints. However, it should be possible to design a language that can quantify all the important constraints. Important constraints are those that are related to site security and those that are related to payment for use of a library. It would be desirable that the language be extensible to support other constraints. Thus, to restate an earlier assertion, extensibility is an overriding design factor. Considering that processor performance and network bandwidth are steadily increasing, extensibility is a good feature to have which will not cost as much as designing and deploying a new system.

Obviously, the language must be flexible enough to allow accurate specification of the requested library in a query. This is one of the main research problems for the proposed work. Another item to note is that there is a high probability that, as active code travels through the network, previously traversed nodes will have the required active network libraries. Given this, the query language or overall query system must support the query of previous nodes along the path that a given active code program travels.

There are three cases in which the resolution service may fail. The first, and most obvious, case is if the library never existed, at least for public access. The second case is if the library no longer exists on the network. An example of the second case is if the library has been expired by the official source. The third case is more subtle. For example, a set of libraries could exist that have the same name, same function, same context, and same primitives. A library can be obtained; however, that library may not work with the active code. There may be any number of errors in the library that may cause the active code to fail. For example, the case where user restrictions are not satisfied is not considered a failure. In the context of this research, successful resolution is defined to be that if a library exists that matches the given parameters, the location of the matching library is returned to the source of the query. No guarantee that the library will work with the requesting code is implied.

3.2.6 Registry Issues

The registry is the system upon which the resolution of active libraries depends. There are two methods to create a networked registry. The first is to maintain multiple, or replicated, copies of the registry. The second is to maintain independent and distributed registries and send queries to multiple servers.

The replicated registry has relatively constant and fast resolution times. Any given server on the network knows what any other server knows, which means that library searches are simple. The cost of this is that all library data must be transferred between every server, which has a significant impact on bandwidth overhead. Another cost is that resources on each server are consumed for duplicate data. A number of replication problems also exist, which include the elimination of the transfer of duplicate information, the frequency at which the information must be sent, when and if the information should be expired, and the actions that are taken when a server is determined to be down. The problems with replication are which node does the updating and how is the information transferred from a single node to potentially hundreds of thousands of nodes.

Unlike the replicated registry, the distributed registry has numerous scaling problems that result for the search process. With a distributed registry, a given node on the network will not know the exact location of the necessary data. Thus, each registry must be queried until the desired result is found. A failure to find a result does not mean that the result is not in the distributed registry — it could mean that the attempt took too long. A simple method to query distributed registries, whose location may or may not be known, is to use the expanding-ring multicast search [92]. The procedure is to multicast a request to the network with a local IP hop count. The hop count restricts the scope of the request to nodes that are a certain distance away from the requesting node. If no response is received, then the hop count is increased and the request is made again. This procedure is repeated until a response is received or the entire network has been searched. One result of the expanding-ring multicast search is that search requests should be kept to only a few hops from the source of the request. This reduces the amount of unnecessary traffic on the network and the likelihood of source implosion. Source implosion is the situation where a node receives more responses than it can handle. This reduction can be

obtained by the use of network caches, which significantly reduces the scaling problems associated with a distributed registry.

Before the choice of solutions is presented, the registry data is analyzed. There are two types of information contained in the registry. The first type includes the more or less static library sources and the second type includes dynamic library sources. A library source is where a resolution service client can obtain the requested library. The static library sources can be considered permanent, and possibly official, sources for a library and this information should be distributed across the network as fast as reasonably possible. The speed requirement is to reduce the unlikely condition that a new library has been just made available after a user on the network desires to use that library. Updates on the static library sources will probably be infrequent. Proper handling of dynamic library sources is the critical design problem. Dynamic library sources include cache servers, user nodes, and switches that happen to have the library. Dynamic library sources are, in general, considered to be caches. The population and content of dynamic sources are expected to change rapidly as caches expire and libraries are replaced. Storage of static source information is more suited to the replicated registries approach, while the dynamic source information is more suited towards the distributed registries approach. However, if the dynamic information is only kept on a "local" basis, the amount of data change across a global network is significantly reduced.

One critical problem for distributed searches is that the registry cannot be organized hierarchically. Other than the absolute library name, there is no enforceable hierarchy in a set of protocol libraries. And if an absolute name is used all the time, there is no purpose for the system. As discussed in Chapter 2, there are a number of problems with absolute names. The reason why a hierarchical organization is preferred is that it significantly reduces the search problem for distributed search registries. Hierarchical distributed searches are used successfully in the Domain Name System [76, 77].

The replicated registry has two serious problems. The first is the overall administration of the global registry. This is embodied in the observation that it may not be possible to develop a globally acceptable policy on, for example, when libraries should be removed. The second is the design of a scalable system with low overhead for situations where there are only a few servers to situations where there are hundreds of servers. These two problems are replaced by one problem in a distributed registry, that of the scalability of distributed searches. The existence of dynamic sources is a problem for replicated registries and is not a problem for distributed registries as dynamic sources can be considered cache servers. The multicast search mechanism, used in the distributed registry, also solves the previous node query problem, as outlined in Section 3.1. The previous nodes would be successively included in the expanding ring. For a replicated registry, a separate mechanism must be developed to solve the querying of previous nodes. Based on this, a distributed registry is used.

One benefit of a replicated registry is that it cleanly solves a problem that was mentioned in Chapter 2. This is the problem of providing the resolution service across different types of internets. Since the registry is replicated, regardless of the internet, all servers know the location

of correct active library. To obtain the library, the server must have access to the correct internet protocol, which the server can obtain from the network. Because the server node has a copy of the registry, some node using the internet protocol must have a copy of the correct active internet protocol. Policy restrictions may prohibit this, but a solution to this policy problem is not addressed in this research as this is considered an administrative matter. The distributed registry can support this internet protocol resolution through the use of query translation. However, these features are not investigated by the experimental system.

3.3 Experimental System

The previous sections provide a general approach to solve the problem of active library resolution. To show that this approach is indeed a working one, an experimental proof-of-concept system was created. This system is discussed in more detail in Chapter 4. The experimental system is not a complete system as it does not provide every feature necessary for an active network to be created. Nor does the experimental system implement all possible strategies that are discussed above and in Chapter 4.

Showing that the approach works is just the first step. The second step is to show that the approach is "good" according to the requirements discussed in Section 3.3.2. One of the major problems in the evaluation of a distributed service is accurate measurement of its characteristics. For example, a network cache is highly dependent on usage characteristics. There is no existing active network architecture, production extensible operating system, or agreement on a central set of features for mobile software, so, obviously, reliable usage characteristics cannot be obtained or generated. The general approach to evaluating systems such as this is to deploy it and, every few years, correct the problems that occur. However, simulation studies, analytical arguments, and comparisons to similar existing systems provide a feasible evaluation approach for initial deployment. These studies, arguments, and comparisons are presented in Chapter 6.

The objective of this research is to develop a proof-of-concept experimental system that will perform active library resolution. An active library is defined to be a set of active network code that is organized into a library for use by other active network code. The experimental system is limited to only those features that are necessary to show that the active library resolution scheme works. These features include an active transport protocol and an active library resolution protocol.

An operating environment for an active network has not been created by this research. The resolution handler is implemented as a TCP/UDP daemon and supports the basic active transport protocol and an active library resolution protocol that supports the query language. The daemon is able to access both the local system and distributed registry of libraries. The daemon is able to insert and execute standard UNIX binaries in the standard UNIX operating environment. The UNIX binaries are viewed, without the loss of generality, as simple active programs. The daemon may not be replaced or removed. The daemon does not provide security checks nor does it track symbols of locally installed active code. ASN.1 data type translation is not supported in the proof-of-concept system. Payment, security, and internet protocol translation features are

defined in the specifications but are not implemented. The demonstration application is a simple ping program that requires a simple checksum library. Note that the demonstration application is merely transported across the network and is not a true active application since it does not copy itself across the network. To allow the application to copy itself across the network, a substantial portion of an active network must be implemented. Showing that a transported program can find its required libraries is the same as showing that an active program can find its required libraries.

The ultimate evaluation is for the resolution service to succeed in obtaining the correct library, as discussed in Section 3.2.5. However, other metrics and criteria are also applicable. Metrics are measurable features of the system and criteria are somewhat subjective and not directly measurable features of the system. These metrics and criteria result from the design objectives as stated throughout this chapter. The metric requirements are low overhead, low resolution times, and scalability and the criteria requirements are extensibility and support of policy constraints. These requirements lead to a robust system. These metrics and criteria are primarily measured against the active library resolution protocol since the protocol is the primary vehicle for the resolution of active libraries. These metrics and criteria are discussed in Chapter 6.

3.4 Summary

This chapter provided an overview of the active network model that was assumed to be present. In this model, the ideal operating environment within a network node was discussed. Critical features and requirements of the proposed active library resolution service were presented. These features include an active library representation, an active transport protocol, and an active library resolution protocol that includes a query language. Metrics and criteria include low resolution times, low overhead, scalability, extensibility, and support of various administrative and policy constraints. These requirements are detailed in Chapter 6.

The overall goal of this research is to investigate a method that resolves active libraries from any arbitrary node in the network. This requires the definition of the node operating environment and the network environment for the active libraries and resolution system. Most importantly, how the information about the active libraries is distributed through the network must be understood. A specific strategy for the resolution of active libraries is discussed in Chapter 4. A demonstration application of a simple network ping program is used to show that the strategy works. The program requires a checksum library, which is obtained from the network, to perform the actual ping.

Chapter 4. Active Library Resolution Service Design

This chapter presents design requirements and critical aspects of the specifications for the resolution service. As discussed previously, there are a number of primitives that must be provided by the operating environment and each active program. The type and function of these requirements are presented in this chapter. After the environment is defined, the features, criteria, and metrics for the active transport protocol (ATP) and the active library resolution protocol (ALRP) are reviewed. ATP provides active code delivery and retrieval while ALRP provides the registry query support. More details about the protocols may be found in Appendices B and C.

4.1 Basic Primitives

A number of uniform access and service functions, or primitives, must be defined for an active network. These primitives must provide access to operating environment resources, other active handlers, and user processes. Especially important are the correct abstractions to let the various software modules query each other for new interfaces. These primitives are only presented to illustrate the requirements for the operating environment. They were not implemented in the proof-of-concept system as they are not necessary to prove that the resolution approach works.

4.1.1 Required Active Code Primitives

For the operating environment to be able to call an active program, the environment must either start execution of the active program at a fixed location, at a fixed symbol, or at some symbol denoted by the transport protocol. Considering that the initial functional parameters must be the same for all active programs, this research defines a number of fixed symbols that must be present in all active programs. Note that active libraries, which are called by executing code that knows which symbols to use, do not have to provide any of the fixed symbols. Another requirement for an active program is that there should be some standard mechanism for the operating environment or other active programs to inform it of changes in the operating environment. Considering the above discussion, the following initial primitives are defined, in the C language, for active programs. These primitives were not implemented as they require an active network operating system and active applications.

```
void start (void);
```

`start ()` is the execution point after the active program is loaded. Active code that is not a library must provide this function. Note that active programs that are designed to be operating system handlers should register to receive system messages in this function and exit the function immediately.

```
int reg (void *handle, char *function);
```

The `reg ()` function allows other active code to register message handlers with the program in question. `handle` is the memory address of the registered function and

`function` specifies the name of the registered function. Operating system handlers are required to provide a registration function even if the function always returns a failure. The `reg ()` function is an example name and format — each message handler call will be slightly different for each operation and should not be forced into a generic call.

```
int dereg (void *handle);
```

The `dereg ()` function performs the opposite of the registration function and has the same operational requirements as `reg ()`. `handle` is the memory address of the registered function. The `dereg ()` function is an example name and format.

4.1.2 Operating Environment Primitives

This section discusses an incomplete set of primitives that should be present in an active network operating environment. This set is incomplete because proper abstractions of data dependent primitives, such as routing tables, are still under investigation [7, 10, 19]. Since this research is not focused on active language constructs and operating environment support for active networks, only the absolute minimal set of primitives are defined. None of the primitives were implemented in the prototype system. This core set of primitives includes memory management, process locking, interface access, and handler management. Related higher level primitives include socket access, compiler and interpreter interfaces, generic network buffer management, dynamic library handlers, and permanent store access. The initial primitives are defined in Appendix A.

A potentially useful environmental feature is to allow multiple instances of the active code to be present. Multiple instances would not be a requirement as it requires a multi-threaded environment. If the environment provides multi-threading, then the appropriate primitives will be present. Thus, the environment must allow querying of primitives and features that are present.

4.1.3 Active Library Resolution Service Primitives

There are a number of service primitives that the resolution service must provide for use by active programs. These primitives must support library resolution as well as allow active programs to implement portions of the query system themselves. For library resolution, two different query mechanisms must be present. The first is to allow the active program to provide a library name and set of interfaces and have the resolver formulate the query. Additionally, primitive interfaces must be available to set restrictions on the query. The second is a mechanism to allow the active program to formulate a query request using the active resolution query language. For query system implementation, a number of functions are required. The first is, obviously, access to the registry information. Other functions include access to the transport mechanisms and a method to instruct the resolver to send certain transport and active program requests to the new query system. Specifications of the required primitives are in Appendix A.3.

4.2 Protocol Design

One classic tradeoff in protocol design is the level of complexity necessary to achieve a certain level of performance. More complex protocols tend to result in large, error prone, and network insecure implementations; however, complex protocols tend to achieve higher throughput for specific conditions. If the protocols for an active network are to be used in a hardware implementation that, for example, bootstraps the operating system for a network device, then the protocol must be simple and small. The typical solution to the tradeoff is to design the protocol suite to operate reasonably well in either environment. Another method, which is the preferred method in this research, is to develop specific variants for each different environment.

One important consideration in the design of a new system is how to group functionality. The transport of active code has been separated from the resolution of active libraries as they are independent tasks. One protocol that can handle both tasks is the ideal solution; however, the multicast requirement of the resolution task and the reliability requirement of the transport task result in an overly complex protocol. The remainder of this section discusses the requirements of each of the two protocols.

4.2.1 Active Transport Protocol

The active transport protocol (ATP) is the protocol used to deliver and retrieve active code. Because ATP merely transports active code, it is not necessary to define a version of ATP that works for all internets. Variants of ATP are defined for each specific internet protocol used by ATP. This section discusses some design issues regarding the operation of ATP. ATP is not the primary research focus and is not the ideal design for use in production active networks. ATP, however, is a test vehicle for illustrating what information must be delivered in active network headers and what minimum functionality is required in the transport protocol for an active library resolution strategy.

The first issue in the design of any internet protocol is whether or not reliable transport is required and, if it is, how reliability will be achieved. For ATP, another issue is designing a protocol that is relatively internet independent. Ideally, ATP should operate with little to no modification over distinct internet protocols such as IP and IPX. Internetwork protocols are generally unreliable in nature and may have different minimum and maximum packet sizes. Packet size issues are discussed in the following paragraph. As discussed in Chapter 3, the ideal design of ATP would be as a new internet protocol. However, this is unrealistic because of the time it would take to standardize and adopt a new and radically different protocol. This leads to the next best case, that of ATP being an integral part of an internet protocol, such as an IP header. The operation of ATP within IP would be to include selected ATP header information within the IP header information. Some modifications to the maximum IP header length may be necessary. ATP within IP allows active code to be transparently included within an IP packet. Because internet protocols cannot assume reliability, ATP must also make the same assumption. It would also be desirable if ATP could operate, with reasonable efficiency, over a reliable transport protocol such as TCP. This, however, can also be specified as a variant of ATP.

When designing for a internet transparent protocol, packet size issues further complicate the situation. Using IP as an example, the best-case maximum IP payload size is 65,515 bytes. However, the IP datagram size is not the end of the issue since the maximum IP size is generally greater than the maximum data size that can be transmitted across the underlying network. Thus, IP datagrams must be fragmented into units that are, typically, under 1,500 bytes in length. IP specifies that at least 576 bytes must be supported by any medium that carries IP, which allows for 64 bytes for headers and 512 bytes for data payloads. Transmission of fragmented datagrams at the IP layer and on directly attached networks generally has a minor performance effect. However, transmission of fragmented datagrams across congested internets can severely degrade performance. Thus, some transport layer fragmentation must be supported by ATP, as it is obvious that most program code will be significantly larger than 512 bytes. To simplify the design and to improve efficiency, a restriction of one request at a time is imposed. Additionally, the common operating mode of ATP is expected to be point-to-point transmission of active code and header information, which means that IP-level fragmentation should not have a significant impact. The result is that a simple automatic repeat request (ARQ) design is employed in ATP to achieve reliability.

The typical method to deal with transmission across different network protocols is to encapsulate one packet format into another packet format and tunnel it through each different internet. However, a better solution for an active network, and for ATP, would be to transfer ATP packets over each internet in that internet's native format. That is, the ATP packet is removed from the payload of one internet packet, say an IP datagram, and copied into the payload of different internet packet, say an IPX packet. In theory, the code itself would perform the copy operation and the translating system may also cache the library being copied. The fundamental problem that prohibits direct copying is, again, the packet size. If packets can be collected and refragmented by some translator device, then transport of data from one internet protocol to another internet protocol can be achieved. This is not as expensive as it may seem since ATP is restricted to one request per packet. ATP is not designed for raw data throughput; rather it is designed to simply and reliably deliver active code.

For the purpose of system demonstration, ATP is specified and implemented for operation over TCP. This avoids all fragmentation and reliability issues. In the future, other variants should be defined for other protocols. Transport of active code is a required function for a resolution system; however, the actual process of the resolution, which is performed by the active library resolution protocol, is the focus of this research.

4.2.2 Active Library Resolution Protocol

The active library resolution protocol (ALRP) will use multicast services to perform library queries. As with ATP, the reliability issue must be examined for ALRP. It is likely that a query for a library will not fit within one packet, which implies that a reliable multicast protocol must be designed. Reliable multicast protocols are complex and can suffer from a number of scaling problems [93-99]. The most important problem is that of multicast source implosion, where a multicast source is overwhelmed by too many replies. This is a crippling problem if every node

on the network is listening to the query — potentially thousands of nodes must reliably obtain the request. A number of solutions may be used to reduce the implosion problem. The first solution is to ensure that the queried library fits within the first packet, which means that a node can automatically discard the packet if it does not have a library with the same generic name. This is because successive packets will only provide additional constraints. The second solution is to restrict, as much as possible, the multicast queries to local nodes. However, the best solution is to remove or reduce the reliability requirement. What allows this to occur is the fact that a multicast expanding-ring scheme is used, meaning that the same packets are sent out in a repetitive fashion and nodes nearest to the source will see the request more than once. Additionally, if the number of packets is small, a simple negative acknowledgment scheme may suffice. The research shows that it is sufficient to only continuously transmit packets, similar to the BOOTP protocol [100], until a response is received. The precise approach is discussed in Section 4.4.3 and Chapter 6. Once the query is received by a server, then the servers would send, via a reliable transport mechanism, information on the matching library to the client. The server would then let the client determine if the library was the correct one.

Table 1. ATP Payload Types/Action Types

Payload Type	*Meaning*	*Description*
0	Ack/Nack	The Ack action provides acknowledgment information for transmitted data. The payload flags distinguish an acknowledgment from a negative acknowledgment (Nack). Applicable only for unreliable variants.
1	Ping	The Ping action is used to determine the remote implementation and the level of support present on the remote node.
2	Get	The Get command is used to retrieve one specified active code.
3	Put	The Put payload type is used to transfer one specific active program or active library. Repeated Put actions must be used to transport multiple modules of code. Note that a Payload Flag must be set for Put to execute the transmitted active code.
4	Remove	Remote clients and servers may wish to delete active code from nodes on the network. This can be accomplished be using the Remove action and transferring the identifying headers.
5	Response	The Response action returns status information about the request and any required data.
6	Status	The Status command forces a status response from the remote node for an outstanding request. Applicable only for unreliable variants.
7	Abort	The Abort payload type requests that the remote node abort an outstanding request. If the response to the supposedly aborted request is received after an abort is issued, the client must ignore the response. Applicable only for unreliable variants.
8	Redirect	The Redirect action informs a client that it should access a particular node to perform the request.
9	Quit	The Quit command terminates a session.
10	Dictionary	The Dictionary action is used to transfer compression dictionaries.
11-199	Reserved	These set of actions are reserved for future use.
200-255	User Defined	These set of actions may be defined in a specific application. Ping requests are used to determine the type of server at the remote end.

4.3 Active Transport Protocol

As discussed previously, the active transport protocol (ATP) may have multiple different specifications, each for a variety of different internet protocols. However, all protocol varieties fall into two general classes of protocols, unreliable and reliable. This section discusses the design of the reliable class. The unreliable class, which was not evaluated, is discussed in Appendix B. Both classes use the same headers, given in Appendix B, and action types, shown in Table 1.

ATP must also be extensible and this means that the following standard rule, paraphrased from [101], must apply — if an ATP implementation does not recognize a received protocol option, it must ignore it. The remainder of this section discusses the reliable variant of ATP, the compressed header scheme, and the naming scheme.

4.3.1 Reliable Transport

The reliable transport variant of ATP is simple and is based on HTTP [11]. The reliable variant uses ASCII text, with special compression rules, to perform all protocol transactions. The format, specified using the augmented Backus-Naur Form (ABNF) [102], of a request line is given below with an example following. The ABNF rules may be found in Appendix B. An Extension-Token is a new token that is added after the specification has been published.

```
ATP-Request          = Action-Type [Name] "ATP/1"
  Action-Type        = "Ping" | "Get" | "Put" | "Remove" | "Response" |
                         "Redirect" | "Dict" | Extension-Token
  Name               = Active-URL | Name-String
```

```
GET active://www.visc.vt.edu/active/libip.active ATP/1
```

Cryptographic data must appear immediately after the ATP-Request line and applies to the MIME headers only.

4.3.2 Header Representation and Processing

As discussed in Chapter 3, the headers used to support active code transport are based on MIME/RFC 822 headers. These headers serve dual purposes. The first is to properly identify the features of the library to the recipient. The second is to serve as registry information. The use of MIME header formats allows active code to be safely transported and executed in a number of environments. This includes electronic mail and the World-Wide Web, which both use MIME headers in the transport protocol. Thus, it would be possible to create a Web browser or electronic mail reader that will execute active code upon receipt, similar to the functionality of the IBM Aglet [52] system.

The major problem with MIME representation is that it is inefficient and utilizes a significant amount of network bandwidth and computational resources. An additional problem is that

MIME representation requires reliable transport as the headers will not fit in the typical network maximum transmission unit of 1,500 bytes. However, the portability, extensibility, and the fact that it is a well understood existing standard are overriding factors that lead to use of MIME in this work. Because text headers significantly increase the amount of overhead incurred by ATP, a form of compressed MIME headers are introduced. The compressed header format will predefine a set of binary values and formats for various headers while allowing new text headers to be introduced. The compression will also help to increase the scalability of the system by reducing the retransmission and processing delays associated with multiple packets. This system has been used in [85, 103]

Table 2. ATP Dictionary Translation Table

Rule	*Encoding Value*	*Rule*	*Encoding Value*
<CRLF>	<0>	Interface-Primitive	<149>
Extension-0	<128>	Composite-Type	<150>
Active-Version	<129>	Interface-Dependence	<151>
RFC1123-Date	<130>	Encrypted-Header	<152>
Vendor-Name	<131>	Encrypted-Data	<153>
Name-Generic	<132>	Content-MD5	<154>
Name-Specific	<133>	Payment	<155>
Name-Specific-Original	<134>	Payment-Accept	<156>
Pragma	<135>	Usage-Cost	<157>
"active"	<136>	Restriction	<158>
"pragma"	<137>	Reserved	<159>-<254>
"priority"	<138>	User-Defined	<255>
Accept	<139>	Extension-1	<128> <0>
Content-Transfer-Encoding	<140>	"Program"	<128><1>
Content-Length	<141>	"Interpreter"	<128><2>
Content-Type	<142>	"Compiler"	<128><3>
Platform	<143>	"Library"	<128><4>
Revision	<144>	"Handler"	<128><5>
Code-Author	<145>	"Device"	<128><6>
Create-Date	<146>		
Revision-Date	<147>	"Predict"	<128><7>
Signature	<148>	"Statistic"	<128><8>

MIME employs a header label followed by header values. Every label and value is a string of ASCII characters, or a token. These tokens can be numeric values or text string identifiers. There are two types of identifiers, registered and unregistered. Registered identifiers are registered with some naming authority such as the Internet Assigned Numbers Authority (IANA) as in [104]. As is standard practice with mail headers, unregistered extension identifiers should start with the character "X." All headers use the ASCII character set as defined in ANSI X3.4-1986 [105]. An example header is given below.

Active-Version = "Active-Version" ":" 1*DIGIT "." 1*DIGIT

The Active-Version header is required in all active code transmissions in any active transport mechanism. The current version is "1.0." The result of the above example rule is given below.

```
Active-Version: 1.0
```

This header would take a total of 20 bytes, 18 for the text and two for the line termination. In the compressed form, this would be represented by four bytes. The first byte is searched for in a dictionary reference to a set of headers, as given in Table 2. The value for the first byte would be the binary integer ‘129’. A table entry and known format must be registered for every header that is compressed. The second byte would be the binary integer ‘2’, which is the size of the header data. Compressed headers are limited to 255 characters in length. The next byte would be the binary integer ‘1’ and the last byte would be the binary integer ‘0’.

The compression technique takes advantage of the fact that the 8-bit US-ASCII used in MIME only uses the lower 7-bits. If the most significant bit is set, this denotes a compressed header. This results in 126 possible encoding rules plus one extension rule. Compressed headers must not be used in implementations that do not claim to be Active-Version 1.0 compliant. If the escape sequence is not present, regular full-text headers are sent, which allows an easy method for extension. Another possible extension header method is to define the Vendor-Name header and then use the set of table entries reserved for user-defined variables. The first extension rule is used to provide an extension to the dictionary table. The compression rules are not used when raw data, e.g., octet-streams content types, are being transferred.

Note that there is an issue of backwards compatibility with the dictionary lookup. For example, an older server application that uses version “1.0” of the dictionary will not understand some dictionary values for a “1.1” application. To rectify this, the “1.0” application ignores the values it does not understand. Additionally, the “1.1” application can “Ping” the server to determine the level of application support and use extension queries as necessary.

4.3.3 Naming Rules

As discussed in Chapter 3, there are three types of names, the original and absolute source library names, the current and absolute library name, and general library names. All three library names, along with revision information, are required to be present in any active code sent across the network. The original name is given by the first source of the active code. The current name is a relative name, given by the current node that has the active code. The generic name is the common name of the library, which is given by the first source of the active code. The absolute library names are represented by the network service type “active,” as defined below with an example following.

Active-URL = (“active:” “//” Host [“:” Port] [Absolute-Path]) | (“none:” User-String)

```
active://www.visc.vt.edu/active/libmine.active
```

The Internet Service Location Protocol [75] can be used to determine which ATP variant should be used. This research uses the ".active" extension as the preferred extension to denote active libraries. The current and absolute library name is encoded into the Name-Specific header, as defined below with an example following.

```
Name-Specific              = "Name-Specific" ":" Active-URL

Name-Specific: active://www.visc.vt.edu/active/libmine.active
```

The original and absolute library name is encoded into the Name-Specific-Original header, as defined below with an example following. The Name-Specific-Original header may not be modified by any node which receives the active code with which it is associated.

```
Name-Specific-Original   = "Name-Specific-Original" ":" Active-URL

Name-Specific-Original:
active://www.ee.vt.edu/active/libmine.active
```

The "none" service type is an important special case. This reserved service type means that the library does not have a specific source on the network and, thus, can never be placed in the registry. The none service type may have some non-unique user provided identification information associated with it, as shown below.

```
Name-Specific: none:"This an example use of the none type"
```

As presented in Chapter 3, general library names have an additional set of rules. These are the application of a context operation to the name and a scope limit on the context and name. The default context is the null context and the default scope is global. Note that the scope is really an additional context operator. The general library name is represented by the Name-Generic header type and component tokens as defined below, with an example given later. The Name-Generic header may not be modified by any node that receives the active code associated with the header.

```
Name-Generic               = "Name-Generic" ":" Name-String
Name-String                = ("MediaType@" SubType) | ((Token | Quoted-String)
                             ["@" Context-String ["@" Scope]])
  Context-String           = Token | Quoted-String
  Scope                    = "IANA" | "ISO" | "IEEE" | "ANSI" | "Global" | "Site" |
                             "Subnet" | "IP" "=" IP-Hop-Limit | Domain-Set |
                             Extension-Token
```

The reserved "MediaType" library name is used to denote active programs that are interpreters and compilers for a particular active programming language. Note that the commercial-at, '@', is a reserved symbol and may not be used in library names. Obviously, the Context-String is an ASCII string that represents the library name context. The Scope header is a set of ASCII

strings that define various registration bodies, such as IANA, or a network scope such as an IP hop-count or domain name. An example Name-Generic header is as follows.

```
Name-Generic: "libip"@ip4@IANA<CR><LF>
```

Note that if the token is in quotes, then it is treated as case sensitive in any operation. Otherwise, it is case insensitive. The compressed form, which is 19 bytes long compared to 31 bytes, is given below.

```
<131><32>"libip"@ip4@IANA<0>
```

4.4 Active Library Resolution Protocol

The active library resolution protocol (ALRP) is used to perform queries of the distributed registry of active libraries. ALRP uses two well-known multicast groups that all servers and each requesting client must join. One group is used to process queries and the other group is used to process announcements about new libraries. ALRP also uses a slightly different operational model from that presented in Chapter 3. This model is more efficient and more precise for the operation of the distributed registry. The primary difference is the use of expanding-ring multicast searches for the library resolution, as opposed to direct queries of Figure 1.

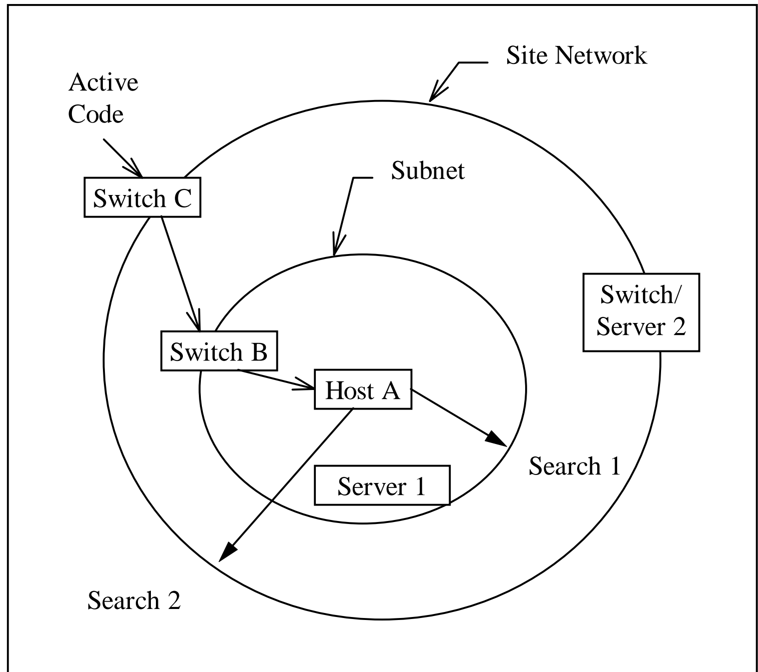

Figure 4. ALRP operational model.

An example network is presented in Figure 4. Nodes C, B, and 2 are switches, while nodes A and 1 are standard hosts. Nodes 1 and 2 are also library servers. In this model, the local directly attached network consists of nodes B, A, and 1. Nodes C, B, and 2 are on a site network that is a

neighbor to the original subnet. Assume that some active code is sent through the network from node C to B to A. Node A is missing some library, so it must make a search request, which is labeled "Search 1." This is a multicast request to the directly attached subnet. Note that library server 1 and switch B, which transmitted and presumably executed the active program, have been contacted in the first search request. Assume that nodes B and 1 do not have the requested library. Host A will then make another search request with a larger multicast scope, i.e., using the expanding-ring multicast search. This is labeled "Search 2." Note that switch C and library server 2 have been reached, in addition to switch B and server 1. If these hosts do not have the library, then the ring is expanded further. If the library is found, the source for the library or the library itself is unicast by all nodes that have the library directly to host A. This set of multiple responses may cause a problem known as multicast source implosion. That is, host A receives more responses than it can possibly process. Multiple responses also have the effect of unnecessarily consuming bandwidth.

Server 1, which is the subnet library server will see all the requests of node A and other local nodes. It can use this information to determine which libraries it should get and cache. Server 1 will not, however, see the responses. Server 1 can also respond to node A with a redirect to itself, which will cause the model in Figure 5 to be used. The only difference in this model is that server 1 performs proxy requests for host A. This is more efficient as it allows the server to see the source of the remote library and cache the retrieved information. This model also directly supports hosts that are on an isolated private network.

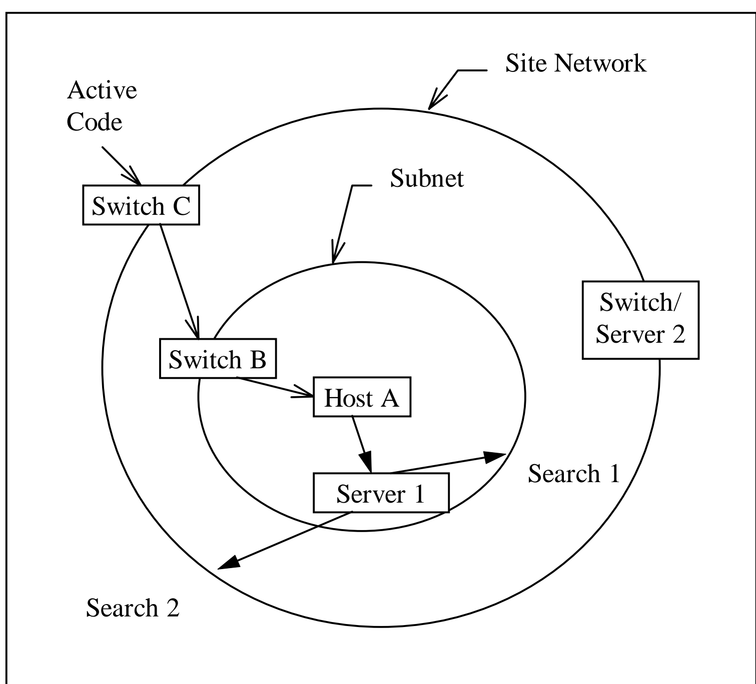

Figure 5. ALRP operational model with Server 1 as a proxy.

ALRP also allows servers to send library announcements to all nodes on an internet. A separate multicast group is assigned for new library announcements. ALRP, like ATP, also has a redirect

mechanism to instruct a client to use a different server. This redirect mechanism is only used by proxy-servers. The remainder of this section discusses how library resolution works, the mechanism for signature resolution, and provides the basic semantics of the query language.

4.4.1 Library Resolution

The process of library resolution is to combine the dependencies, or the unresolved function primitives, and the library name into a query that is sent to an ALRP server. The query can also have additional constraints, such as obtaining a library only from a specific site. The query constructs form a query language that must be extensible since the information used for library resolution is extensible. A problem quickly results if the registry format changes but the query language cannot change to search the new information. Ideally, registry format changes should not result in a modification of the query language syntax. Another important design consideration is the policy constraint requirement. Policy constraints can be viewed as restrictions, which are a set of exclusions from the search, and allowances, which are a set of inclusions in the search. Constraints and basic queries are performed by applying an and operation on the set of constraints.

The example that is used to demonstrate the language properties is a hypothetical *libsocket.active* library. This library implements the standard POSIX socket library which typically consists of twenty or so functions. Since *libsocket.active* only provides high level socket access, another network protocol library must be obtained. This hypothetical library is called *libactive.active* and exports twenty or so interface routines for socket libraries to use. Note that *libactive.active* uses a number of fixed operating environment calls that do not need to be exported or resolved over the network.

Assume that some active program needs to use *libsocket.active*. The program knows the generic name of the library and builds the request. The program may choose to add some constraints and then should provide the set of library interfaces to match. The safest match would be to match the library name and all twenty odd functions. However, chances are that the active program will not use all twenty functions. There may be a more efficient version of *libsocket.active* available that does not implement all twenty functions but implements the functions that the active program uses. The set of dependent interfaces used is the application developer's choice. This results in the requirement of strict and loose matching capabilities for the query language. Once this library is obtained, the resolver handler determines that *libactive.active* is required by *libsocket.active*. The same procedure would be used to obtain *libactive.active*.

4.4.2 Signature Resolution

A signature is defined as the symbol name and data type parameters of an exported primitive function. External interface signatures are resolved using the Interface-Primitive header. Functions that are internal to the active code do not need to be externally resolved and should not be specified as an external interface. Signatures must be unique in each code module. Signature resolution is the strategy used to match the requested signatures and the actual signatures.

Correct signature resolution is defined as an exact match of a set of primitives to another set of primitives. As discussed in Chapter 3, signature matching across different programming libraries requires the data types to be translated into a common data type representation. The recommended representation is ASN.1.

There are a number of considerations for proper signature resolution. Each function, within a code fragment, has one unique symbol that identifies it from all other functions. The function may have zero or more return values and zero or more parameters. The return values and parameters may be comprised of primitive data types or complex data types. Furthermore, for pre-compiled binaries, each language has different calling conventions. All of the above requirements must be present in interface definition, as shown in the Interface-Primitive definition below. Some mechanism for dealing with complex data types must be present as well; this is performed by the Composite-Primitive header. An Extension-Set is simply a set of Extension-Tokens. The C-Interface-Parameter and the C-Composite-Type rules provide specific definitions for the C programming language, as is discussed in Appendix B.

```
Interface-Parameter      = "Internal" | C-Interface-Parameter | "@"
                           Composite-Name | Extension-Set

Composite-Primitive      = "Composite" ":" Sub-Type ";" Composite-Type ";"
                           Composite-Name ";" *(Interface-Parameter ",")
  Composite-Type         = C-Composite-Type | Extension-Set
  Composite-Name         = Token | Quoted-String

Interface-Primitive      = "Interface" ":" Sub-Type ";" Symbol-Name ";"
                           [Return-Value *("," Return-Value)] ";" [Passed-Value
                           *("," Passed-Value)] [";" Call-Type]
  Symbol-Name            = Token | Quoted-String
  Return-Value           = Interface-Parameter | Extension-Set
  Passed-Value           = Interface-Parameter | Extension-Set
  Call-Type              = "C" | "Pascal" | Extension-Token
```

Conversion from function declarations to the interface representation is performed by iterating over the primitive data types in order of appearance. The procedure is illustrated via a simple example. This example uses the following simple C function declaration.

```
void function (struct example, float *return);
```

The `example` composite data type is formatted as follows.

```
struct example{
     int a, b;
     float z;
     struct example *next;
}
```

The composite primitive data structure, using the Structure-Primitive rule, would be referenced as follows.

```
Composite: C; example; int, int, float, @example *
```

The proper Interface-Primitive statement would then be as follows.

```
Interface: C; function; void *; @example, float *; C
```

Note that if one was seeking to match the primitive function on a C source level, the C call parameter passing value should not be specified. For example, the definition or search primitive request would be as follows.

```
Interface: C; function; void *; @example, float *
```

4.4.3 ALRP Queries

As discussed in Chapter 3, the basic mechanism to make scalable queries in the distributed registry is to move the data as close as possible to the query originator. This can be accomplished by having localized servers that cache registry requests which are obtained from the network. Ideally, the local server would be a proxy for the client in question, though a number of different operating modes are possible. Because of the network efficiency of a proxy-server that also caches, this is the mode of operation used in the proof-of-concept system. There are two active library retrieval methods that can be used. The first method is to let the requesting client retrieve the protocol itself and the second method is to let the server send the protocol. ALRP does not, by itself, provide a facility to transport the protocol; however, the design of the system allows ATP, or another protocol, to be used for either of the two methods.

In many respects, the resolution service is similar in nature to obtaining a file, in this case, a library, from FTP servers. Multiple copies of the files are located throughout the network with no systematic pattern. There are certain FTP servers that are well-known archives of files, but there is no mechanism that automatically searches all FTP servers. The closest search service for FTP is “archie” [106], a global index database of the contents of a large set of anonymous FTP servers. It allows searches based on the name of the file or a text description of the file, if available. Queries in the archie service can take up to several minutes and only return the location of the server. The primary differences between archie and the proposed system are that 1) archie is a replicated database, rather than a distributed database, 2) this system allows more precise description of the characteristics of active libraries than archie, 3) unlike archie, this

systems allows the libraries to be retrieved or automatically sent by ATP to the requester, and 4) in general, this system is much more flexible.

The method used in the proposed system is to have library servers on the network that are not globally indexed. A global index has the effect of being a replicated registry and can be implemented separately from the proposed system. As discussed earlier, to search the network globally consumes excessive resources and time. Ideally, libraries would be obtained from the nearest local server. Thus, clients can send requests to a local server that will proxy requests for the client. The local server will search other remote servers and return the library, if found. The library and the locations found are stored in the server's cache, so when another client makes a similar request, the results are retrieved from the local server. This is similar in concept to Web proxy-servers [107]. As noted earlier, accurate characterization of the performance of such a system is impossible without long-term deployment and known library content. Caching is not studied in this work.

Two possible protocol variants were studied. Both variants use expanding-ring multicast to transfer the data in a set number of rounds. At the end of each round, each protocol waits for a specified time period to allow receivers to process and return a response. The first variant employs an error control scheme after all expanding-ring multicast rounds are ended. The error control scheme, which uses negative acknowledgments, has each receiver transmit a list of missing packets back to the source. Then the source multicasts only the collective missing packets to all nodes. The second variant continuously retransmits each round at the maximum TTL until a response is received or a timeout is reached. Both variants are evaluated in Chapter 6.

To keep resolution times low, three methods may be used to fill the server caches. These methods are library announcements, proxy-servers, and search request observations. Library announcements are used by remote servers to "prime" caches throughout the network. Proxy-servers allow client hosts to make requests through a server, which may cache the requests that are made. Observation of search requests is similar to a proxy-server except that all library search requests received on the network are used to determine if the cache should be pre-loaded.

The use of proxy-servers allows the creation of what this research terms translation servers. Translation servers are proxy-servers that reformulate a query request, translate the ATP or ALRP request from one internet protocol to another protocol, and perform many other computations on the requests. Translation servers were not implemented or investigated in this work.

4.4.4 Active Registry Query Language

As with ATP active code transport, the query language uses a form of compressed statements. Each query is for one library only, which adheres to the one request nature of ALRP and ATP. Query statements are based on the previously defined ATP headers. The semantics of the original design for the query language were boolean. The current design performs an and

operation on all query statements, combined with regular expressions searches. Regular expression searches replaces the semantics of the not operation. The or operation is not supported as it does not make much sense for a library query service. If needed, the or operation can be achieved through the use of multiple queries. Removing the or operation also has the benefit of reducing the protocol complexity for hardware implementations. Specific rule definitions are in Appendix C.

For example, to find and have the server send the *libip* library of Section 4.3.3, with the constraints of the Solaris operating system, the revision level "1.0" or higher, within two to four dollars per unit cost, and two interfaces, the full text query would be as shown in Figure 6. Note that the line numbers in the figure are not part of the actual query. Using the compression rules, the size of the query can be significantly reduced. Compression rules were not investigated in the proof-of-concept system.

Understanding the formulation of the query is important to understanding the mechanics of the resolution service. Each query statement must be a separate line of text and defines a match against the headers that are associated with an active library. The first line defines the search target. Normally, the search target will be a Name-Generic statement, but the search target can be for any ATP header. Note that only one search may occur for each query request. After the search statement, any number of optional pragma statements may follow. After the pragma statements, any number of search constraints may be added in any order. The first two PRAGMA statements in Figure 6 are optional and are instructions to the server. The first statement defines the number of times to try the search and the second statement instructs the server to use an ATP Put payload type to return the result. Alternatively, the number of seconds to wait can also be specified. A number of server instructions can be used, such as the maximum number of items to return and the maximum number of servers to search, and all such instructions must occur immediately after the search statement, or the first line.

```
1     Name-Generic: "libip"@ip4@IANA
2     PRAGMA RETRY TIMES 10
3     PRAGMA RETURN-TYPE SEND
4     0Platform: .*-.*-Solaris-.*
5     2Revision: 1.0
6     6Usage-Cost: 4.00, US-Dollars
7     3Usage-Cost: 2.00, US-Dollars
8     0Interface: C; socket; int; int, int, int; C
9     1Interface: C; bind;
```

Figure 6. Example query request for an active library.

Table 3. Query Categories

Categories	*Description*
0	Regular expression match
1	Regular expression not matched
2	Greater than numeric match
3	Greater than or equal to numeric match
4	Equal to numeric match
5	Less than numeric match
6	Less than or equal to numeric match

Constraints are added to the search by seven categories, as shown in Table 3. The first two categories specify the use of pattern matching through the use of regular expressions. Specific forms of regular expressions are discussed in Appendix C. The remaining five categories define simple numerical tests on the header. Thus, each header that a numerical test is applicable to must only have one number and that number must be the first field present in the header. The date rules are the only allowed exception to this case.

Line number 4, in Figure 6, indicates that the preferred platform is some "Solaris" operating system device; however, any platform is acceptable. Note that ".*" tokens are regular expression tokens that are used in between the hyphens to denote any match for this field. The next line indicates that any revision greater than "1.0" can be used. Lines 6 and 7 indicate that the cost must be within or equal to $2.00 to $4.00. Note that restrictions are processed strictly in order of reception and conflicts are resolved in a manner specified with each header. For example, if a cost restriction of greater than $4.00 was sent and then a cost restriction of greater than $6.00 was sent, the restriction that should be used for costs is the widest range, which means that the resulting restriction would be $4.00 or higher. The last two lines define the primitive interfaces that must be present or not present. The first primitive, `socket`, must exactly match whereas the second primitive, `bind`, only requires a partial match and cannot be in the library.

4.5 Summary

This chapter presented the environmental and protocol requirements and an overview of initial results. The operating environment needs a set of defined hardware access routines and each active program needs to provide a set of known initialization routines. This provides the standard basis for the execution of active code. To deliver and retrieve the active code, a specialized protocol, the active transport protocol (ATP), is defined. Another protocol, the active library resolution protocol (ALRP), provides mechanisms to query a distributed registry through an expanding-ring multicast search. Both protocols use a unique compressed MIME format to reduce overhead and increase processing efficiency. Support for intermediate processing devices, such as translators, cache servers, and firewalls, can be easily added to either protocol.

Chapter 5. Experimental and Simulation System

To show that the active library transport and resolution concepts can work, a prototype implementation was created. Testing of this prototype, which supports all important ATP and ALRP features, is discussed in this chapter. To verify that the proposed active library resolution strategy can work in large internets, a simulation system was built and studied. The discrete event simulator works on hierarchical networks and should provide a good prediction of ALRP performance. This chapter discusses the implementation of both the prototype and the simulator.

5.1 Experimental System

The implementation of the prototype was sufficient to show that the library resolution scheme works, within the constraints discussed in Chapter 3, and to obtain experimental measurements. The following sections discuss the features implemented and describe how the prototype was verified. Experimental measurements are discussed in Chapter 6.

5.1.1 Implementation Status

The prototype supports both ATP and ALRP. The reliable ATP implementation supports the ATP Get, Put, Post, Remove, and Ping methods. The Dictionary method is not implemented as compression support for ALRP is not implemented. ATP header transport supports both compressed and regular text headers, as well as the DLL-based dynamic addition of new headers. As discussed in Chapter 3, the headers for the policy and cost constraints are not implemented. All the headers can be searched using the regular expressions that are available on Solaris [108]. Note that this is not the regular expression mechanism that is specified for use in ALRP. ALRP regular expression rules are discussed in Appendix B. The ATP implementation supports multiple libraries and processes but does not support multiple requests within one connection. All libraries are translated into a locally unique namespace, which means that duplicate libraries are eliminated. To support the execution of pre-compiled binaries, all libraries are put into a well-known index file. Pre-compiled binaries must include special code to search this index file. Automatic resolution of libraries that are in the Interface-Dependence header of pre-compiled binaries is implemented. Pragma statements are not supported. All other aspects of ALRP, except for the dictionary compression of interface headers, are supported. Thus, the library announce mechanism, the proxy-server mechanisms, and the query mechanisms are implemented.

5.1.2 Protocol Verification

To verify that the implementation works correctly, a number of tests were conducted. The prototype should be able to handle most any, if not all, possible combinations of resolution requests from different nodes. These tests included various types of simple regular expression matching and various situations. The most complex test was to verify that the proxy-server functions work. This test sends a modified BSD *ping* application [109], as discussed in Chapter

3, to a destination node, or client. This *ping* application requires a library which means that the client must perform a network search to obtain the library. The search will be forced to a proxy-server which will obtain the library for the client.

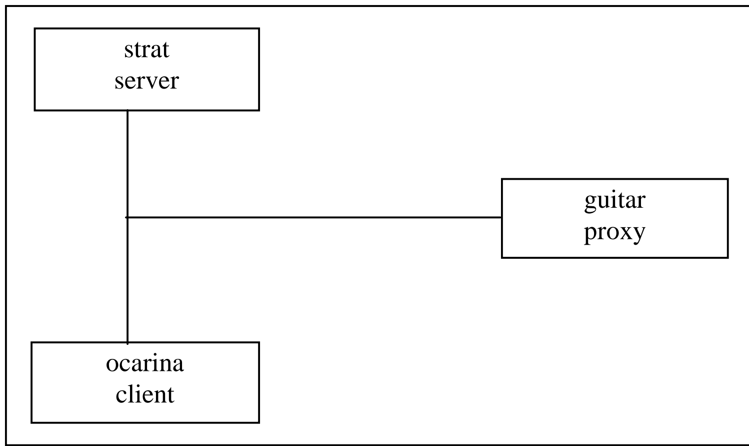

Figure 7. Configuration for verification test.

The configuration of Figure 7 was used for this test. In this network, node ocarina was the destination of the *ping* application and node strat was the server that had the *libping.so* library required by the *ping* application. The *libping.so* library was pre-loaded onto strat. Node guitar was the proxy-server and was also used as the node to initially transfer the ping application. Note that the source node of the ping application is immaterial. All nodes had to be on the same network as the New Engineering Building multicast services were disabled at the time of the test. The expanding-ring search was verified through separate tests performed previously when multicast services were enabled.

```
1    Name-Generic: ping
2    Name-Specific: active://strat/home/dlee/atp/strat/ping
3    Active-Version: 1.0
4    Content-Type: active/binary
5    Vendor-Name: VT Hula Hoops, Inc.
6    Platform: Solaris-All-All-All
7    Revision: 1.0
8    Code-Author: David C. Lee (dlee@vt.edu)
9    Create-Date: Mon, 1 Jun 1997 23:21:12 GMT
10   Revision-Date: Tues, 1 Jan 2000 0:0:0 GMT
11   Test-Header: This is a uncompressible header.  Does it comes through?
12   New: Test new header.
13   Dependence: Name-Generic: libping.so
```

Figure 8. Headers for the ping application.

The headers for the ping application are shown in Figure 8. Headers for the *libping.so* library are similar and are not shown. The line numbers are not a part of the transferred headers. These

headers are compressed and decompressed during the transfer. Lines 1 through 10 and line 13 are headers provided for by the ATP specification in Appendix B. Lines 11 and 12 are new headers to verify the extensibility of ATP headers. The significance of line 12 will be discussed later.

Figure 9 shows the trace output for guitar, Figure 10 shows the trace output for ocarina, and Figure 11 shows the trace output for strat. Unnecessary trace information has been removed and line numbers added to help in the discussion.

```
1   atp post ocarina ping ping.hdr
2   Executed 10 _hdr_new ht 180
3   srv (27277): Content-Length: 24696
4   srv (27277): (10) Done transferring data.
5   ...
6   srv (27278): Connection from 128.173.92.98
7   srv (27278): Request processed.
8   srv (27278): Sending redirect.
9   srv (27278): 224.100.100.100 a multicast address.
10  (27278) SENDING: sec 880506802 usec 552950
11  (27278) SENDING: sec 880506802 usec 556340
12  (27278) SENDING: sec 880506802 usec 559015
13  ...
14  srv (27278): (10) Answer received.
15  srv (27278): Received URL: (69) active://strat/home/dlee/atp/strat/code/tmp005450
16  srv (27278): Waiting for data.
17  srv (27278): code file /home/dlee/atp/guitar/code/tmp027278
18  srv (27278): hdr file /home/dlee/atp/guitar/hdr/tmp027278
19  Executed _header_new
20  srv (27278): Expected size is 3773
21  srv (27278): (11) Data transferred.
22  srv (27278): First search string = [Name-Generic: libping.so]
23  srv (27278): first [Name-Generic: libping.so]
24  srv (27278): Match found.
25  srv (27278): Sending response (71 bytes) to 128.173.92.98:1024.
26  ...
27  srv (27279): Processing command Get /home/dlee/atp/guitar/code/tmp027278 880506642 ATP/1
28  srv (27279): (10) Sending file /home/dlee/atp/guitar/code/tmp027278.
29  Executed 10 _hdr_new ht 180
30  srv (27279): Content-Length: 3773
31  srv (27279): (10) Done transferring data.
```

Figure 9. Proxy-server (guitar) trace for verification test.

Lines 1 through 5 in Figure 9 show the proxy-server transferring the ping application to the client. The corresponding lines in Figure 10 are lines 1 through 7. Note that Figure 10, line 3 indicates the local name of the *ping* binary. Line 2 for guitar and line 5 for ocarina show the execution of a special header. These lines reappear throughout the verification tests and correspond to the header at line 12 of Figure 8. This header is not part of the ATP specification

but instead is a new header used to show that header specification is dynamically extensible. The importance of this demonstration is discussed in Chapter 6.

```
1    srv (14546): Processing command Post 880506802 ATP/1
2    srv (14546): proc_cmd: Post [880506802 ATP/1]
3    srv (14546): code file /home/dlee/atp/ocarina/code/tmp014546
4    srv (14546): hdr file /home/dlee/atp/ocarina/hdr/tmp014546
5    Executed _header_new
6    srv (14546): Expected size is 24696
7    srv (14546): (10) Data transferred.
8    srv (14546): Attempting to obtain dependency.
9    srv (14546): First search string = [Name-Generic: libping.so]
10   srv (14546): first [Name-Generic: ping]
11   srv (14546): 224.100.100.100 a multicast address.
12   (14546) SENDING: sec 880506641 usec 910672
13   srv (14546): First search string = [Name-Generic: libping.so]
14   srv (14546): first [Name-Generic: ping]
15   ...
16   srv (14546): Redirect received to 128.173.92.128.
17   srv (14546): Waiting for response.
18   ...
19   srv (14546): Received URL: (71) active://guitar/home/dlee/atp/guitar/code/tmp027278
20   srv (14546): (17) Getfile started.
21   srv (14546): code file /home/dlee/atp/ocarina/code/tmpa14546
22   srv (14546): hdr file /home/dlee/atp/ocarina/hdr/tmpa14546
23   Executed _header_new
24   srv (14546): Expected size is 3773
25   srv (14546): (17) Data transferred.
26   srv (14546): Code type active/binary
27   srv (14546): Executing /home/dlee/atp/ocarina/code/tmp014546
28   PING strat.visc.ece.vt.edu (128.173.92.91): 56 data bytes
29   64 bytes from 128.173.92.91: icmp_seq=0 ttl=255 time=2.363 ms
30   64 bytes from 128.173.92.91: icmp_seq=1 ttl=255 time=1.370 ms
31   64 bytes from 128.173.92.91: icmp_seq=2 ttl=255 time=1.291 ms
32   ...
```

Figure 10. Client (ocarina) trace for verification test.

The next step is for ocarina to search for the dependency expressed on line 13 of the headers shown in Figure 8. This particular dependency results in a one packet ALRP request and ocarina's attempt to resolve it runs from line 8 through 25 in Figure 10. Lines 9 and 10 show ocarina's search attempt on ocarina itself. Because the library is not present on ocarina, ALRP is used and a request is sent via an expanding-ring multicast search, as shown in lines 11 through 15. Since guitar is a proxy-server, it observes the request and issues a redirect to ocarina, as shown in lines 6 through 8 in Figure 9. Ocarina's reception of the redirect and subsequent wait for a response from guitar is shown on lines 16 through 18 of Figure 10.

```
1    srv (6162): Connection from 128.173.92.128
2    srv (6162): First search string = [Name-Generic: libping.so]
3    srv (6162): first [Name-Generic: libping.so]
4    srv (6162): Match found.
5    srv (6162): Sending response (69 bytes) to 128.173.92.128:1024.
6    ...
7    srv (6163): Processing command Get /home/dlee/atp/strat/code/tmp005450 880506802 ATP/1
8    srv (6163): Transferring file.
9    Executed 10 _hdr_new ht 180
10   srv (6163): Content-Length: 3773
11   srv (6163): (10) Done transferring data.
```

Figure 11. Server (strat) trace for verification test.

Guitar then makes a request for the library itself as seen in lines 9 through 13 of Figure 9. Strat was setup to ignore library requests from any node that was not a proxy-server. Strat receives and processes the request, as shown in lines 1 through 6 of Figure 11. Guitar obtains the library from strat for its own cache, as shown in lines 14 through 21 of Figure 9. The corresponding transfer of the library from strat is shown in Figure 11 lines 7 through 11. Guitar verifies the reception of the library in lines 22 through 24 of Figure 9. Guitar now must send a response message, shown on line 25 of Figure 9, to ocarina indicating where ocarina should obtain the library. Normally, the client should obtain the library from the proxy-server. Ocarina obtains this response and fetches the library from guitar, as shown in Figure 10 lines 19 through 25. This corresponds to lines 27 through 31 in Figure 9. The library location is displayed at line 19 in Figure 10.

The last step is for ocarina to execute the ping program with the shared library. The shared library location is stored into the well-known index file at `/tmp/anos_db.txt`. This file is used by the ping program to load the shared application and execute it. The execution process is shown in lines 26 through 33 in Figure 10. As can be seen, the ping application is pinging the host strat.

5.2 Simulation System

As discussed previously, the simulation system is used to characterize the performance of ALRP over large networks. The precise evaluation strategy is discussed in Chapter 6. The system is a discrete event simulation that is implemented using a simple state machine. Three modules are used in the system, a module for the source node, the server nodes, and the router nodes. The simulation system uses fixed transmission times for an Ethernet frame and the interframe gap. It also incorporates processing delays and protocol timeouts to obtain resolution times. How the simulation networks are generated, each node type, and system operation and limitations are discussed in turn.

5.2.1 Network Generation

The networks that are generated for simulation are hierarchical in nature, as shown in Figure 12. This is similar to the backbone topology of the Internet, which is a loose hierarchical collection of service providers. Each level of the network has a set of routers that are interconnected to routers on higher levels. The number and destination of these links are random. There are no interconnections between networks on the same layer. This configuration creates a multicast tree in which the top-most network is the root, or core. This structure also matches the model for spare mode multicast routing [110, 111]. As multicast traffic dominates unicast traffic, this is a good model. Each network has a set of randomly allocated nodes that are connected to a router on the network. The networks that have end system nodes are referred to as level 0 networks, or leaf networks. All other networks are transit networks. Networks of one hop away from the level 0 networks are referred to as level 1 networks, networks of two hops away from the level 0 networks are referred to as level 2 networks, and so forth. This is also shown in Figure 12, with squares indicating nodes and ovals indicating routers. There is only one router per network. The number of levels and number of transit networks at each level vary depending on the evaluation criteria and metrics. There are always more level 0 networks than level 1 networks, more level 1 networks than level 2 networks, and so forth.

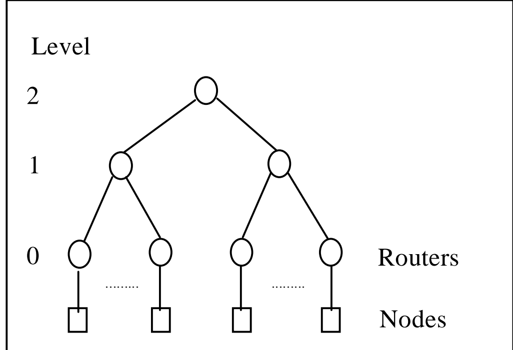

Figure 12. Example hierarchical network.

The user may select a number of parameters, including the number of possible library servers, the maximum width of the network, the number of levels, the number of nodes, and the number of networks.

5.2.2 Source Node Operation

The algorithm for the source node is given below. Note that the simulator has a much more complex algorithm to support the error control design. The algorithm shows all states required for ALRP using continuous retransmissions, as implemented. The states for the error control design are not used and, thus, not listed. These designs were briefly discussed in Section 4.4.3 and are discussed in Chapter 6. Also note that the simulator does not support proxy-server or caching functions.

1. If the source needs to send a packet, multicast it to all local nodes and the router.

- If it is not the last packet of the request, schedule the next packet to be sent.
- Otherwise, wait for the user-specified delay.

2. If user-specified delay occurs, increment the TTL and schedule the first packet of a new round to be sent.
3. Check to see if a response (announcement) has been received.
   - If it is an announcement, multicast an acknowledgment.
   - Delete all events and exit simulation.

Note that the multicast acknowledgment of step 3, in the simulator, is never processed. This is because the library announcement has been received and the simulation is effectively over. The acknowledgment is used in real systems to instruct receivers to shutdown the process handling this request. The source node will transmit the query, wait for a specified delay based on the number of packets transmitted, increment the TTL, and begin the query transmission anew. This process is repeated until an announcement is received or timeout conditions are reached. Timeout conditions indicate that no response was received. The reason for the delay being based on the number of packets is to ensure that Ethernet capacity is not exceeded. Roughly five rounds of 64 packets will begin to exceed Ethernet capacity. This corresponds to roughly 300 rounds for a one-packet request, which is significantly more than any realistic situation will require.

5.2.3 Server Node Operation

The source node also uses a simple algorithm, as presented below.

1. If a packet has been received:
   - Test for random error loss occurs and discard packet if necessary.
   - If this is packet zero, check to see if we are the library server. If not, ignore all other packets.
   - Otherwise, test to see if we have received all packets. If so and if we are the library server, schedule an announce packet to be sent and a timeout.
2. If an announce packet is to be sent, send it directly to the source and schedule a timeout.
3. If a timeout occurs, schedule another announce packet to be sent.
4. If an acknowledgment is received, shutdown process.

The server nodes will wait until all the packets are received then, if it has the library, the server will send an announcement to the source. The announcement should not be multicast as a large number of these can quickly cause implosion problems. This announcement is periodically retransmitted until the source acknowledges it or a counter expires. Note that, in step 1, if the server does not have the requested library, the server will clearly know this when packet zero is received. Packet zero contains the name of library that is being search for and all other packets contain additional constraints.

5.2.4 Router Node Operation

The router nodes are the simplest to implement and use the following algorithm.

1. If TTL has expired on this packet, discard it.
2. Test for random error loss occurs and discard packet if necessary.
3. If this is a multicast packet, send it to all other outgoing interfaces.
4. If this is a unicast packet, send it to the next destination node.

A test of visited nodes, in addition to the hierarchical tree structure, is used to ensure that multicast packets are never duplicated. The test simply eliminates cyclic packets. Unicast packets have a route stored in their headers and the router simply forwards them.

5.2.5 Operation and Limitations

The simulation system tracks a number of counters and transmission time values at all nodes in the network. These are used for the evaluation in Chapter 6. The system uses receiver-based packet loss when discarding packets. This is a good model of an Ethernet segment as one receiver may properly obtain the packet and another, on the same link, may obtain a corrupted packet. The error model also uses a uniform loss rate for all links, which simplifies the simulation. Because of this uniform, load-invariant loss rate and the fact that there is no other traffic on the simulated network other than ALRP, each node ensures that packet collisions will never occur. This has the effect of extending the resolution time and is a relatively accurate model of what occurs on Ethernet, which uses collision-detection. This system assumes that all nodes have infinite queues. Each node also monitors to see if Ethernet capacity is exceeded. Finally, each node has a random, uniformly distributed zero to 100 ns, processing delay for each processing step.

The number of packets per round, the link error rate, the TTL count of the network, and delay factors can be changed without the need to generate a new network. Both the network generation and simulation system accept user-specified seeds for the random number generator. Verification and validation of the simulation is presented in Chapter 6.

5.3 Summary

This chapter discussed the implementation of the prototype system and tests for proper functionality. All major ATP and ALRP systems were implemented and verified. The verification example included in the dissertation text is a complex proxy-server test that uses a number of ATP and ALRP functions. The result was that the library resolution strategy works as expected. The chapter also reviewed -simulator operation and key features of the system. This included the algorithms used for the source node, the server nodes, and the router nodes. The simulation error model and how the simulated networks were generated were also discussed.

Chapter 6. Results

As discussed previously, there are two important assumptions that were made in the development of this system. The first assumption is that user-level binaries are effectively the same as active code. The second assumption is that an active network operating system is present. One obvious observation is that measurements made with the simulation system are imprecise at best. Regardless, the results should be good indicators that the system should perform reasonably well in a production environment — they simply are not the exact numbers that would be achieved by a production system. This chapter begins with a discussion on the evaluation of ATP and ALRP through the use of the experimental prototype. Then the results of simulation analysis of ALRP performance are presented. The metrics, criteria, and procedures used to obtain these results are reviewed. The expected content of ALRP packets are also discussed. Source data for figures and tables can be found in Appendix D.

The original design of ALRP used an error control scheme to reduce the overhead and provide improved resolution time. Simulation results indicated that this scheme would fail under high loads and that an alternative scheme, using continuous retransmission of the data, would yield better performance and lower overhead. These two systems are discussed in Section 6.3.

6.1 ATP Experimental System Evaluation

As discussed in Chapter 3, there are three metrics and two criteria that are used to determine how "good" the active library resolution system is. The active library resolution system consists of both ATP and ALRP. Of these five metrics and criteria, scalability is not applicable to ATP. The criteria of extensibility and policy constraints are jointly applicable to ATP and ALRP and are discussed with ATP. As discussed in Chapter 4, it is known that ATP is not the best possible design. ATP is loosely based on HTTP and, thus, some of its evaluation methods are based on how well it compares to HTTP. ATP could be implemented as a modification of HTTP, which would increase the amount of header overhead and require key modifications to the HTTP specification. At a minimum, the interpretation of the HTTP methods must be widened and ATP headers must be supported. HTTP would also have to be extended to support protocol translation and other functions. As discussed in Chapter 3, the ideal design for an active code transport protocol is to incorporate it into a network layer protocol such as IP. As a number of other investigators are working on this solution [44, 88], ATP is primarily used as a vehicle to support the resolution system and validate what headers and services need to be in the active code transport protocol. The remainder of this section discusses the evaluation of ATP.

6.1.1 Extensibility

Extensibility is hard to define as it means different things to different people. Regardless of definition, the criterion of extensibility is subjective in nature and typically describes a set of features that help make the system extensible. The literature on extensible systems typically does not define criteria for extensibility; rather, they show how the system is extensible

[17, 18, 112-115]. Some researchers do attempt to define characteristics of extensibility [116-118], but these characteristics are generally applicable only to software design and not to protocol development. Since literature on active library resolution in active networks is not available, this work defines how extensibility can be achieved in an active library resolution service. This definition is based on common design principles of network protocols. Specifically, a set of features, or characteristics, that are needed for extensibility are defined. For a network service, protocols should ensure backwards and forward compatibility with previous versions. It should be relatively straightforward to add a new feature to the protocol, and, more importantly, the extensions should be consistent with other features. "Reliably straightforward" is too vague a concept to be a good characteristic. In this work, new features are new protocol headers. [118] does, however, note that extensible systems should allow the addition of new methods without knowledge of the system internals. Thus, extensibility is defined to have the three characteristics of compatibility, consistency, and only require knowledge of the external protocol specification.

As discussed in Section 4.3.2, a method for determining the level of ATP support is defined. This method provides for a mechanism to ensure backwards compatibility with "official" ATP extensions and vendor-based extensions. Consistent extensions to this specification require detailed knowledge of the headers and the application of engineering discipline. The specification of ATP, in Appendix B, defines well over 20 different headers. This does not include the number of unrelated MIME headers that have been standardized. Thus, the format and specification of MIME headers are well-understood and maintaining design consistency to these headers should not be a serious problem. Additionally, regular expression searches work as long as the header format is consistent across all implementations — the header itself does not need to be consistent with any other header. The numerical tests will work if the first field specified in the header is a number. Thus, the query system is also extensible.

It must be noted that MIME is a mechanism for defining headers and not a language per se. Thus, one can view MIME as having ultimate extensibility as anybody can defined any header that they desire. This, obviously, is a poor mechanism for extensibility. However, MIME also has the property of requiring a number of specific headers to be supported. Thus, these standard headers can be viewed as giving MIME language-like features and allow MIME to be extended in a structured manner.

The final criteria for extensibility is to require implementers to only have knowledge of the external specification. Thus, implementers should be able to add new ATP headers through selected application programming interface (API) calls at run-time. Using dynamically linked libraries, the experimental system was designed to show how new headers could be dynamically inserted and processed by the system. The validation of this system is discussed in Chapter 5 and shown in Figures 8 through 11. The extension header, "New" was added to the system. The use of this header can be seen in the headers of Figure 8 on line 12. This header is processed and the user-provided code is executed in line 2 of Figure 9 and line 5 of Figure 10, as well at other times during the verification test. This shows that a production active network implementation, with full ANOS facilities, could easily and painlessly extend itself.

6.1.2 Overhead and Response Time

Overhead is defined as additional data that is not necessary for active code execution but is needed by the resolution service protocol. In other words, overhead can also be measured as the percentage of the total data that is not active code data, or, inversely, as efficiency. The major overhead in the proposed system is the transport protocol for active code. A large part of the active code transmission header is considered to be an accepted overhead and cannot be eliminated. The overhead is reduced by using a simple dynamic compression technique, as discussed in Section 4.3. Response time is defined as the time between the reception of request and the reception of a response to the request. Response time is closely related to the amount of header overhead and the amount of actual data transferred.

Table 4. ATP and HTTP Comparison

Library	Data Size (bytes)	ATP		HTTP	
		Response Time (s)	Header Count (bytes)	Response Time (s)	Header Count (bytes)
libstdio.a	400,162	0.704	2,156 (605)	0.608	428
libstring.a	278,232	0.497	1,344 (335)	0.503	429
libsocket.a	24,586	0.195	1,116 (263)	0.220	428

Because ATP is closely related to HTTP, its performance is compared to HTTP. Table 4 shows the result of this comparison. For both systems, the server used background Network File System (NFS) traffic to obtain the libraries. Each data set is the average of three runs. All measurements were taken on the production Ethernet network in the New Engineering Building using the same server and client machines. Parts of the GNU C library [46] were compiled into individual libraries and transferred from a server to a client. The libraries correspond to the functions that are in the standard C header files of “stdio.h,” “string.h,” and “socket.h.” Transferred ATP headers included the interface primitives and some additional information about the library. The interface primitives were defined to match the C header files. Transferred HTTP headers were standard headers sent by the Netscape FastTrack 2.01 Server [119] for binary requests. As can be seen in column 4, the header counts for ATP are significantly higher than the header counts for HTTP, as shown in column 6. Note that the number in parenthesis is the number of bytes saved by the ATP header compression algorithm. An average of 20.6 percent compression is achieved across all three ATP transfers. For two cases, the ATP header overhead is roughly within one 1,500 byte Ethernet packet. One case requires two Ethernet packets. This is comparable to the roughly one Ethernet packet of overhead for HTTP. Note that the header overhead for HTTP and ATP are spread across both the request and response. The response times are also comparable — considering background traffic and differences in implementation, they are nearly equivalent. Thus, ATP performance and HTTP performance are similar.

6.1.3 Policy Constraints

As discussed in Section 3.3.4, policy constraints are additional non-functional requirements that are placed on the resolution system. These requirements are typically administrative and commercial in nature. As it is not possible to conceive of all possible policy constraints, important site security and payment constraints are examined. The major concern with the policy constraint evaluation is that the evaluation must be able to show that the provided set of constraints is complete; however, because of the extensibility of the system, this is not a serious concern. The three major concerns with network data are that the validity and integrity of the data are maintained, that data access must be controllable at the source and destination, and that data usage can be restricted by some cost function. A discussion of how these features are met is used to show that the policy constraint criteria is satisfied.

Data validity and integrity are addressed by the use of cryptographic checksums or digests to ensure both data validity and integrity. This performed by the Signature or the Content-MD5 header to protect the data and by the Encrypted-Header header to protect sensitive headers. The use of the Encrypted-Data header and the appropriate cryptographic algorithm ensures the confidentiality of the data.

The control of data access is partially performed by the Restriction header. Obviously, the client and the server can be implemented so that other access rules are used. The Restriction header simply provides a mechanism for the server to tell the client, or vice-versa, the access restrictions for the requested library. The header allows restrictions based on hostname, domain names, and other extension keywords. The Restriction header can also be used in library searches, which improves the flexibility of the system.

Cost-based data access restrictions must address secure payment services and provide some indication of how much a service costs. Because electronic commerce is a large and new area of research [120, 121] and because the economic model is not known for active network libraries, a simple cost-per-unit model is used in ATP. In this model, every library has a finite cost and users are either willing or not willing to pay that cost. Negotiation is not supported, though it may be possible for server and client implementations to perform some form of dynamic price tag adjustment. Sellers advertise the cost of their libraries and the acceptable forms of payment through the Usage-Cost and Payment-Accept headers. Buyers accept the price tag by requesting the library and specifying how payment will be made. Payment specification is performed by the Payment header. Standard forms of credit card and bank transactions are supported.

Thus, the three concerns of data validity and integrity, that data access, and cost-based data restrictions are addressed by ATP headers.

6.2 ALRP Experimental Prototype Evaluation

The use of a prototype to evaluate ALRP can only show that the concepts of ALRP work and that the performance of ALRP is acceptable on a local network segment. Simulation studies are used

to show that ALRP should have good performance on large-scale internets. This section discusses the prototype results and Section 6.3 discusses the simulation results. This section first discusses the data content of ALRP, which justifies the design assumption that ALRP data sizes will be small. Next, performance results are presented and shown to be close to the results of the simulation.

6.2.1 ALRP Content Analysis

This section discusses a simple analysis of the probable data content that will be used in ALRP. This analysis is used to justify design assumptions presented in the research. A selection of 20 header files were examined for content that may be included in an ALRP query. Additionally, the symbol tables of 26 libraries were analyzed for a similar purpose. The results of this analysis are presented in Table 5 and Table 6. These results are only an estimate that is useful in setting design requirements for ALRP and active libraries.

Table 5. Header File Analysis Results

		Function Name Size (Bytes)				Number of Parameters			
	Count	Total	Average	Low	High	Total (bytes)	Average	Low	High
Function Count	310	3409	11.0	2	30	675	2.1	0	9
Composite Types	115	1065	9.3	2	16	407	3.5	1	26
Totals	425	4474	10.5	2	30	1082	2.6	0	26

The selected files are mostly various network related functions, with the exception of “stdio.h,” “fcntl.h,” and “des_crypt.h.” As the precise content of active libraries is not known, the exceptions are used to “round out” the analysis. The exceptions are also used in many UNIX programs. The other files are socket calls, RPC calls, resolution services, device calls, and calls used in an implementation of a routing protocol. With the exception of the routing protocol files, all files come from Linux 2.1.43 header files [122], patched with various IPv6 support files and other modifications. The routing protocol files come from version 1.3.5A of the Multi-threaded Routing Toolkit [123].

The analysis measured the number of functions per library, the average size of a function name, the average number of parameters that a function uses, the average number of composite data structures that are used by the functions, and the average number of variables in a composite data structure. The standard definition of a UNIX library is the set of “.a” files, which contain a number of modules, or “.o” files. However, in some respects, a “.o” file could be considered a library as not all the modules present in a “.a” file are used by all applications. This distinction is partially reflected in the analysis as the multicast routing protocol functions are found in the same “.a” file. Through the use of dictionary encoding techniques, primitive data types for most all languages can be encoded into one byte. Thus, parameter counts equate to the number of bytes used by each function or composite data type. Function and composite data type names are directly related to the number of bytes they use. Key variables were included in the composite data type analysis. As some data is system dependent, some composite data types were not fully

included. The assumption is that system dependent data can be better determined by matching on the Platform header than the inclusion of additional unnecessary search constraints. Constants and macros, which are only useful for dynamic compilation and interpretation, were not generally included in the analysis — upper bound estimates are that the complete addition of constants and macros would, at most, double the number of functions in the analysis. Constants and macros are generally internal to a library and not exported. However, some constants and macros are exported and it becomes the decision of the library designer on how, and if, to export them.

Table 6. Library Symbol Analysis Results

Description	Total	Unique
Number of libraries	26	26
Number of symbols	5,519	
Number of externally referenced symbols	1,012	663
Number of bytes for externally referenced symbols	9,200 bytes	6,214 bytes
Average number of unique externally referenced symbols per library	25.5 symbols per library	
Average number of bytes per symbol	9.2 bytes	9.4 bytes

The results presented in Table 5 are rounded up to the next integral value. As can be seen, the average function name size is 11 bytes and the average number of parameters per function is three. The average number of composite data types per function is six — this number is obtained by dividing the composite type count of 115 by the 20 header files. The average length of a composite data type name is ten bytes and the average number of variables used by a composite data type is four. About 16 functions and 12 composite data types exist in an average header file. Source data can be found in Appendix D.

Another analysis, presented in Table 6, was performed using symbols in UNIX libraries found on the same Linux system. Symbols in a library can be both functions and externally referenced data types. The second column provides the total data, which includes duplicate entries. The third column provides data based only on unique entries. Note that the average number of externally referenced symbols, that were unique, for each ".o" file was not calculated as ".o" files are commonly duplicated in each library. This duplication also implies that the average of 25.5 externally referenced symbols per library can be taken as a lower bound. This is twice as high as estimates of Table 5 but also includes global variables within a library. The average symbol length of ten bytes is close to the 11 bytes in Table 5.

An earlier estimate, obtained from the same Linux system, shows that the probable worst case will be 100 functions, with at most ten parameters each, per library. The average case was around 25 functions with five parameters each, which corresponds with the above estimates. Function names were at most 30 characters long with the average being around ten. The probable worst case count for the associated composite data types was about 30, but 50 is used in the analysis below. The average case was under ten composite data types. The number of primitive data types that were included in a composite data type ranges from 25 to 50 types, which corresponds with the results in Table 5. If a high level of compression is used, each parameter is only one byte and the name for the composite data type can be indexed with a

one-byte representation. Compression schemes were not investigated by this work; however, a simple mechanism is described in Appendix C. The result of the library analysis is that the variables that control ALRP data size are the function names and the number of functions. The worst case estimate is 100 * (10 + 30), or 3,000 bytes, to describe all exported functions, and 50 * 50, or 2,500 bytes, to describe all composite data types. 1,000 bytes should be more than sufficient to describe any other constraint on the code. The total is 7,500 bytes, which is roughly five packets if an MTU of 1,500 is used, or 14 packets if the minimum IP MTU of 576 is used. For the probable average case, 625 bytes should describe the exported functions and 800 bytes should describe other constraints. The total of 1,425 bytes fits within one Ethernet packet or three minimum sized IP packets. Note that the expected minimum path MTU for IPv6 is 1,280 bytes [124], which means that two minimum sized IPv6 packets will be used. The uncompressed ATP headers used in Section 6.1.2 are, at worse, two Ethernet packets — these headers are representative of what would be included in a library search. The MIT ANTS project [2, 8-9] postulates that active code will be retrieved based on a digital signature of each library and, this research would add, a specific library name. A minimal sized IP packet is more than sufficient to contain this data, which further supports the case that the average number of packets per request should be one or two packets.

Note that ALRP has a maximum of 64 packets, which is sufficient for 35,684 bytes in minimal sized IP packets and 92,160 bytes in a typical Ethernet packet. ALRP has 20 bytes of headers. Roughly 590 uncompressed and individual search constraints can be placed in 35,684 bytes, assuming that each constraint averages 60 bytes. Active libraries that require more constraints must be redesigned.

6.2.2 Experimental Measurements and Simulation Validation

Figure 13 shows the results of the prototype performance measurements and a comparison to the equivalent simulation measurements. The only design evaluated here is the continuous retransmission design as error control is not recommended for use. The rationale for the design selection will be discussed in Section 6.3.4. The network configuration used is one server and one client on two different but directly attached Ethernet segments. Four plots are shown, two each for the simulation and prototype networks. The simulation plots are denoted with a “S” in the legend and the prototype plots are likewise denoted with a “P” in the legend. Each network was configured for one percent and five percent packet loss, which is also shown in the legend. Each data set presented is the average of three random runs. As the maximum number of ALRP packets in any given request is 64, the plot shows the resolution time for increasing request sizes. The prototype measurements are made with three important differences from the simulation measurements. First, the time delay at the end of each round is 0.05 seconds per packet as opposed to 0.03 seconds per packet of the simulator. Second, the prototype client has 0.04 seconds of startup delay due to process creation. Third, the prototype server does not check for packet reception until after all packets are sent for the current round. The last difference is made to support multiple designs in one implementation and the first difference is to account for the process creation overhead. Considering these differences, the prototype results are comparable to

the simulation results. This validates the simulation system for this one case. Other cases require networks that are too large to obtain experimental measurements.

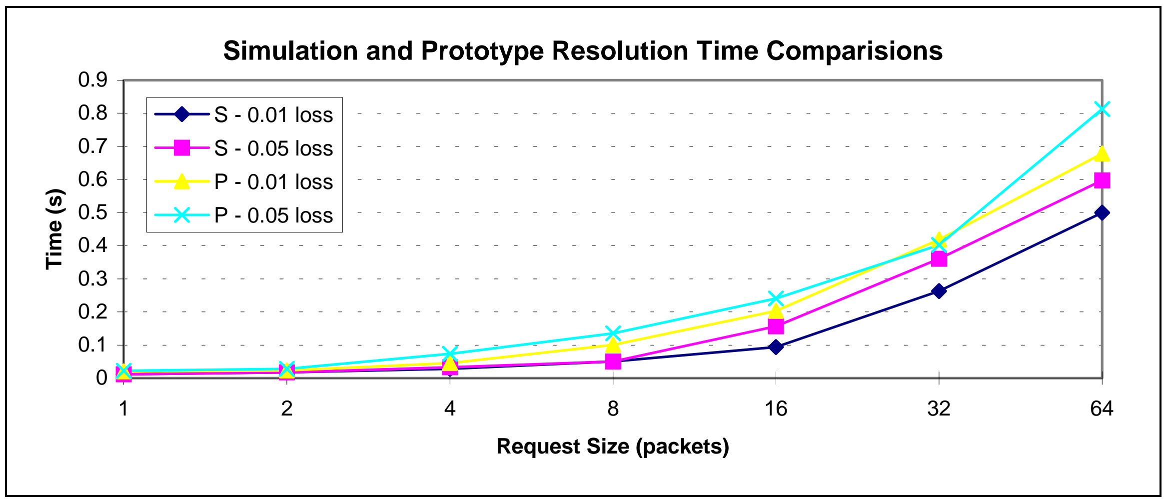

Figure 13. Simulation (S) and prototype (P) resolution time comparisons.

6.3 ALRP Simulation Evaluation

The initial evaluation of a global search protocol like ALRP is typically performed through the use of a simulation tool. Simulation allows the performance evaluation of a global protocol without global deployment. The simulation evaluation is merely a predictor of production operation. There are a number of issues with simulation measurements, most notably the correctness of the network and service modeled. A simple simulation validation was performed, as described in Section 6.2.2. This section describes the configurations simulated, the metrics used, and the initial simulation results. The two designs of Section 4.4.3 are compared in Section 6.3.4. The chosen design is evaluated in Section 6.4.

6.3.1 Simulation Setup

There are three factors that are used in the simulation evaluation. These are packet counts, TTL counts, and packet loss rates. In ALRP, one to 64 packets can be sent per request, which means that packet counts can be used over variations in data size. Note that the assumption used in the evaluation is that the library in question is not available locally. The use of packet counts is equivalent to the use of data size since the packet counts model data blocks of n * 1,440 bytes, where n is the number of packets. 1,440 bytes is the maximum ALRP payload size for an Ethernet network.

TTL count values are used to define network boundaries. For example, a TTL count of one defines the directly attached network. A TTL count of two includes the directly attached network and all immediately connected adjacent networks. The theoretical maximum number of TTL counts and thus, retransmissions, is 255. The typical maximum TTL count for unicast

transmissions in the production Internet is around 15 hops from the source node. This would mean 15 retransmissions will occur. The practice in the current MBONE is to define network boundaries by using TTL counts that increase by powers of two, which means that up to six retransmissions may occur. This research uses the unicast TTL counts as the number of retransmissions.

The end-to-end packet loss rate determines the number of retransmissions required for reliable delivery of the query. For the majority of simulations, the two loss rates of one percent and five percent are used. This is sufficient to study the performance of ALRP under the normal range of error rates. One set of simulations uses loss rates from one percent and 75 percent packet loss. Packet loss higher than 75 percent results in a network that is effectively out of service.

Key factors that are not included or evaluated in the model are the effects of multicast routing, effects of different active languages, effects of different search criteria, effects of caching, and the effects of different database implementations, protocol implementations, and processors. These factors are either approximated by another factor or are really a separate evaluation of a different system.

Another factor for the error control variant of ALRP is the use of multiple retransmission timeout values. Retransmission timeout values have a direct impact on resolution time. The evaluation goal would be twofold, the first part is to determine good retransmission timeout values and the second part is to determine their impact on resolution time and scalability. However, since the error control variant is not recommended for use, this analysis was not performed.

Three metrics, latency, throughput, and the probability of multicast implosion, were studied for use in scalability measurements. Various types of latency can be measured and the most appropriate is resolution time, as discussed in Section 6.3.3. Throughput is a measure of how fast the system operates and is not used in the evaluation approach. Throughput is useful when there is a significant amount of data to be transferred. An average of two packets and at most 64 packets is not a significant amount of data. As discussed earlier, multicast implosion is the case where a source receives more responses than it can handle. Multicast implosion is only a factor for the ALRP error control design and not a factor in the ALRP continuous retransmission design.

The evaluation approach is to model five different possible network sizes, as shown in Table 7 and Table 8, and show that the resolution time for each configuration is acceptable under normal operating conditions and, thus, scalable. Scalability under high packet loss conditions may not be achievable. The group membership size is the number of nodes in the active network that are using the resolution service. This size does not include nodes that serve as routers. The number of routers is equal to the total number of networks. The number of leaf networks is the total number of networks in the system that are attached to end systems. There are an unspecified number of transit networks that constitute the network routing hierarchy. Thus, the total number of networks is the number of leaf networks plus the number of transit networks. The TTL count defines the maximum width of the network by hops through routers. Note that the link count in

Table 8 is the TTL value reduced by one. This link count indicates the number of links between the source and destination, or the diameter of the network. The link count is used to generate the one percent and five percent end-to-end error values for each network.

Table 7. Test Network Characteristics

Case	Group Membership Size	Level	TTL	Number of Leaf Networks
1	10	0	2	2
2	100	0	2	2
3	1,000	1	3	10
4	10,000	2	5	100
5	100,000	4	9	1,000

Originally, the evaluation of six different network configurations were planned. The largest case of 1,000,000 nodes, 10,000 networks, and a TTL of 13, did not simulate as it took more compute resources than were reasonably available. It also turned out that it is not necessary to perform the simulation of the largest case. The reason for this is that the remaining cases have a strong linear correlation and the larger case has an extremely high probability of following the trend.

Table 8. Test Network Error Characteristics

Level	Diagram	Number of Links	Link Error Rate For End-to-End Loss Of	
			0.01	0.05
0		3	0.00334	0.01695
0		3	0.00334	0.01695
1		4	0.00251	0.01274
2		6	0.00167	0.00851
4		10	0.00100	0.00512

Table 8 shows sample network configurations for each case as well as calculations for link error rates for each case. Link errors are assumed to be independent. Given a link error rate, E_{link}, and the number of links, n, the end-to-end error rate, $E_{end\text{-}to\text{-}end}$, is calculated as follows.

$$E_{end-to-end} = 1 - (1 - E_{link})^n$$

Thus, the individual link error rates can be calculated from the following equation, assuming the number of links, n, and the end-to-end error rate is known.

$$E_{link} = 1 - \sqrt[n]{1 - E_{end-to-end}}$$

Simulation data was also obtained for varying loss rates for case 4, which has six hops. The end-to-end error rates and the link error rates used for this data set are presented in Table 9. They are calculated exactly the same as for the values in Table 8.

Table 9. Loss Rates For Error Analysis

End-to-End Error Rate	Link Error Rate
0.01	0.00167
0.05	0.00851
0.10	0.01741
0.20	0.03651
0.35	0.06928
0.50	0.10910
0.75	0.20630

In summary, three random configurations were used for each of the five network sizes. Each random configuration used a seed value randomly selected from the telephone book. One server and one resolution source, randomly chosen, were used in each network. The server was on a network at the maximum TTL count value from the source. The networks, which may or may not be realizable, are modeled based on the assumption that there are approximately 100 nodes per network and the number of hops is approximately the natural log of the number of networks. The scenarios cover networks of different sizes, different loss rates, variations in server and source locations, and variations in factors. Bandwidth for all links is assumed to be a uniform 10 Mbps. Delay and loss rates are assumed to be uniform across all links.

6.3.2 Extensibility, Constraints, and Overhead

Extensibility and policy constraints are applicable to both ATP and ALRP and they were discussed in Section 6.1. The remainder of this section discusses why ALRP overhead is acceptable.

The “typical” range for IPv4 applications is defined from the standard IP specification of the minimal packet size as 576 bytes and that an estimated 64 bytes are used for headers and 512

bytes are used for data [125]. ALRP headers are 20 bytes long, which is much less than the typical average of 64 bytes for each header.

6.3.3 Resolution Time and Scalability

Resolution time is defined as the time it takes from sending a query to receiving the result of the query. The time it takes to transfer libraries depends on the library size and network transmission capacity and is not part of what is referred to as resolution time. There are some questions as to how fast the resolution time should be. From one standpoint, it would be reasonable to expect some delay upon the initialization of a new service. Another observation is that these active network protocols will probably have a long lifetime. Or, for the initial service request, the resolution time could be high. However, over multiple requests, the average resolution time and overall overhead may be negligible. Based on the above, ten or fewer seconds for the initial request may be reasonable. However, users have a history of not desiring to wait and a time around the average Domain Name System (DNS) resolution time of 0.3 seconds [126] would be desirable. Thus, the design goal was to have a resolution time of 0.3 seconds to ten seconds. As can be seen, for most simulated networks configurations, this goal should be easily achievable. Also note that request caching, which should help improve resolution time, is not studied in this work.

A service is scalable if, as the service grows at a linear rate, the performance and usability features grow at a less than exponential rate. As discussed earlier, there are three aspects to scalability in this system, performance scalability, resource scalability, and usage, or namespace, scalability. Performance scalability means that as the network grows in size, the resolution times and overhead do not also grow in an exponential fashion. The evaluation of ALRP performance scalability is discussed in Section 6.4.3. Resource scalability is directly related to the overhead of the registry caching operation and is not an important factor in the distributed registry design. Measurement of namespace scalability is a harder problem as there is no known namespace scalability measurement other than actual deployment and usage. However, because of the allowance of duplicate library names and contexts, as well as the use of matching function signatures, namespace scalability could be considered infinite.

6.3.4 ALRP Error Control versus Continuous Retransmission Simulation Results

As discussed in Chapter 4, two designs for ALRP were evaluated. The first design used an error control scheme and the second design simply continuously retransmitted the data. The analysis results, discussed below, indicate that the error control scheme will fail under high loading conditions. One known architectural modification that may improve the error control scalability is to use intermediate routers to perform error recovery as opposed to only using the source node. This modification was not investigated as it requires radical alterations to existing multicast services and is a separate research issue.

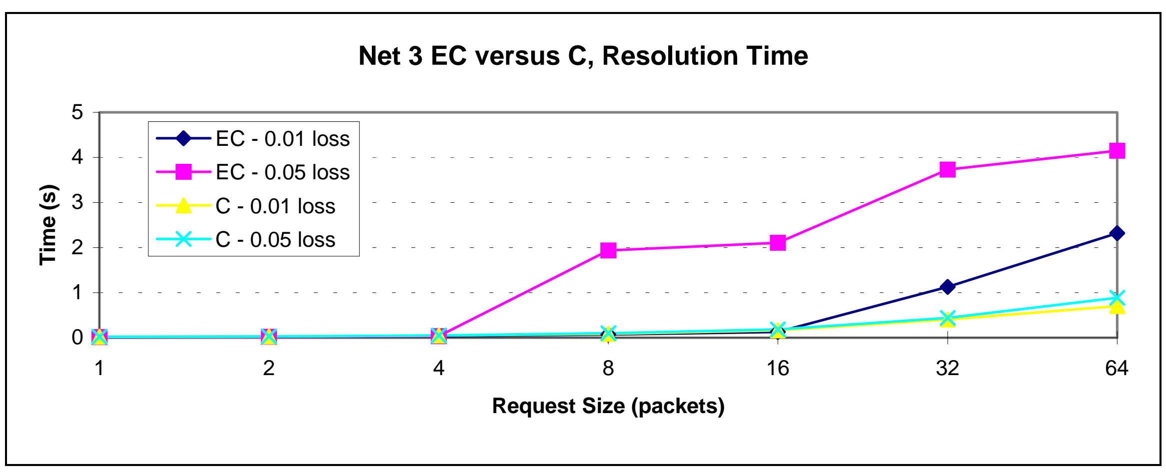

Figure 14. ALRP resolution time for error control (EC) versus continuous retransmission (C) schemes.

Both designs were simulated using the moderately sized network of case 3 of Table 7. The results of the simulation are shown in Figure 14 and Figure 15. Both designs were simulated using one percent and five percent packet loss. All data sets are the average of three runs with different seeds. Figure 14 shows the resolution time for increasing request sizes. The error control scheme, labeled EC, clearly has longer resolution times than the continuous retransmit scheme, labeled C. Additionally and not shown, the EC cases have significant implosion problems for the 32 and 64 packet cases. The EC curves are piecewise linear as they incorporate random backoff and other delay schemes to improve implosion handling. A larger number of runs per data set will help "smooth out" the curve.

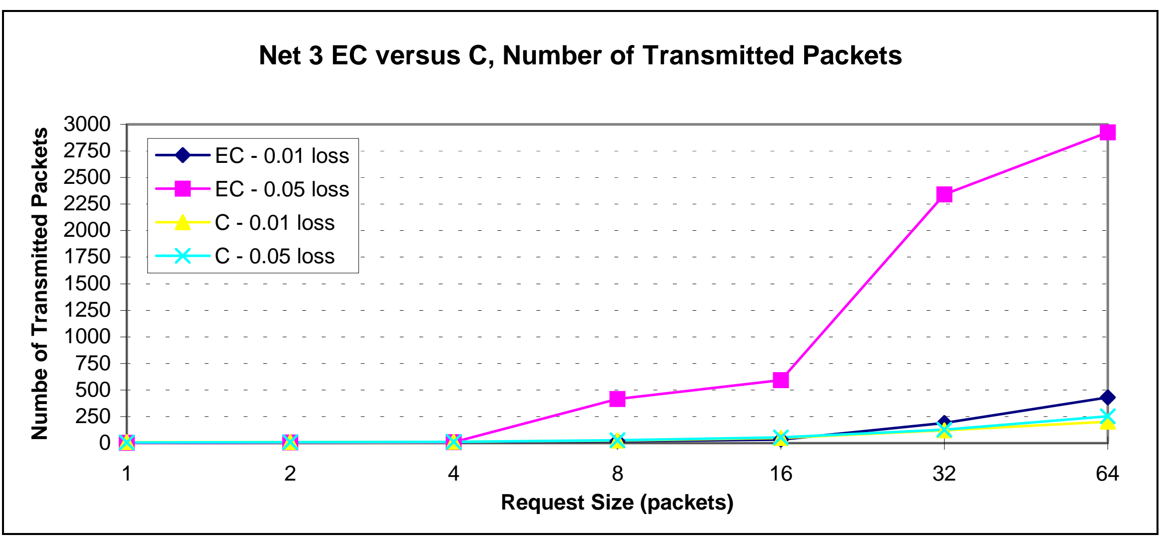

Figure 15. ALRP packet counts for error control (EC) versus continuous retransmission (C) schemes.

Figure 15 shows the number of transmitted packets compared to the number of packets that compose the query. The setup is otherwise the same as for Figure 14. These transmitted packet curves do not reflect any delay scheme and provide a better indication of scalability. For the

same query and network configuration, the number of packets for the one percent packet loss EC case is roughly equivalent to both continuous retransmission cases. However, there is an up to six-fold increase in the number of packets for the five percent packet loss EC case. The five percent loss EC case is clearly exponential and not scalable.

The primary problem with the error control scheme is that with moderate to high error rates and/or high number of responding receivers the error responses that are sent by the receivers start to overwhelm the network. This does not mean that the service will not work. Rather, the resolution time becomes longer and causes significant congestion at certain nodes. This congestion is the result of multicast implosion. Proper setting of timeout values can reduce the implosion problem at a significant cost in response time. The continuous retransmission approach avoids any action by the receivers, which significantly reduces the amount of retransmissions. It is not possible for the continuous retransmission case to cause multicast implosion by itself. For low error rates, the continuous approach is no different from the error control approach as error control is never invoked. The continuous retransmission design does have slightly more overhead for low to moderate conditions; however, the number of extra packets is at most a few dozen. Thus, the continuous retransmission strategy is the proposed design for an active library resolution service. The strategy also has the benefit of simplifying the design and implementation of the system.

6.4 ALRP Continuous Retransmission Simulation Results

As continuous retransmission is the preferred design, the remainder of the results are only for that design. This section presents the resolution time results for the different simulated cases, a performance analysis of ALRP under increasing packet loss conditions, and an analysis of the scalability of ALRP. Data for the charts may be found in Appendix D.

6.4.1 ALRP Performance Analysis

Figure 16 and Figure 17 show the simulation results for each of the five different networks of Table 7. Each network was simulated at one percent and five percent packet loss. The legend describes the network number and packet loss rate. For example, “Net 1 - 0.01 loss” indicates network case 1 at one percent packet loss. Each result is an average of three random runs. Figure 16 shows the resolution time as a function of increasing packet counts and Figure 17 shows the total number of transmitted packets as a function of increasing packet counts. The request size, plotted as the dependent variable, are the number of packets that are in the request. Thus, a request size of 16 indicates that the request took 16 packets to send. The resolution time is the time between the request and the response. The transmitted packet count indicates the number of packets that were sent by the client and all servers during the execution of the request.

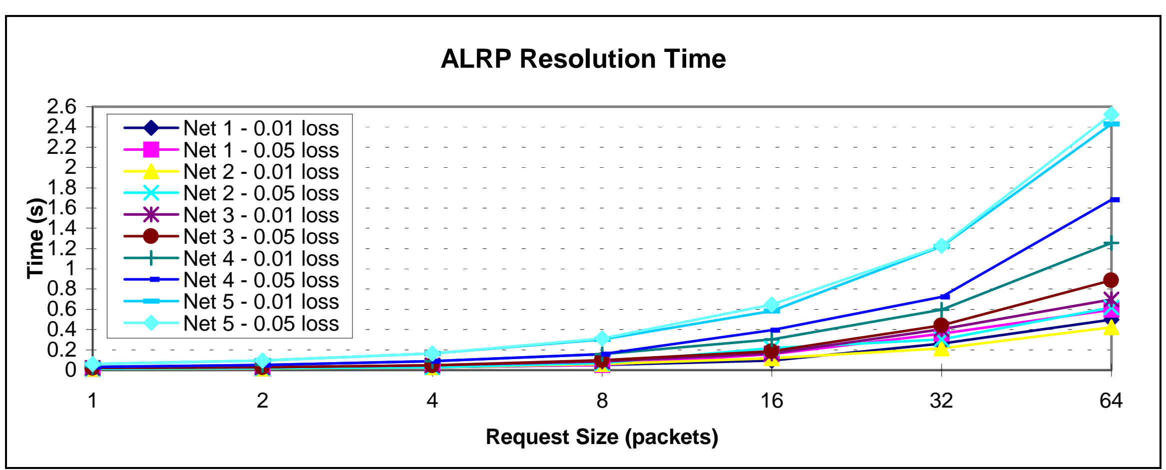

Figure 16. ALRP resolution time analysis for all networks and error rates.

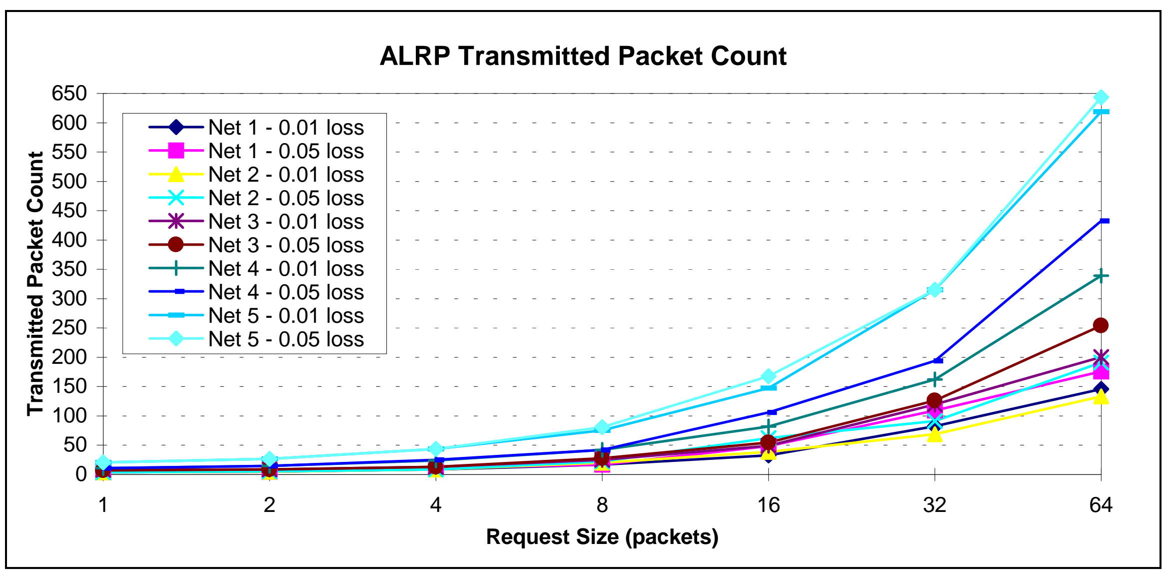

Figure 17. ALRP resolution time analysis for all networks and error rates.

The resolution time for the worse case network size and packet count is 2.52 seconds with 644 transmitted packets. This performance is well within the specification with a marginal impact on the network. For the expected request size of one to two packets, ALRP achieves superior response time of under 0.1 seconds and negligible packet count of under 28 packets for all cases. Note that while the curves appear exponential, this is not the case. Both figures have curves that are based on increasing the size of the network and not increasing the TTL values, which is the better comparison. These plots are presented to understand the protocol's performance in order to determine what type of scaling properties will be present. Individual cases can be isolated and studied. The analysis of Section 6.4.3 will show that the service is indeed scalable.

6.4.2 ALRP Packet Loss Performance Analysis

To study ALRP performance under poor network conditions, network case 4 was simulated under increasing end-to-end packet loss from one percent to 75 percent. This meant that the maximum link error rate was 21 percent, which is high but still low enough to allow packets to be transmitted. The results of this study are presented in Figure 18 and Figure 19. Figure 18 shows the resolution time for each different error rate and Figure 19 shows the corresponding transmitted packet count for each different error rate. The plots show the seven different error rates as a function of the request size and the plots are otherwise the same as the plots in Section 6.4.1.

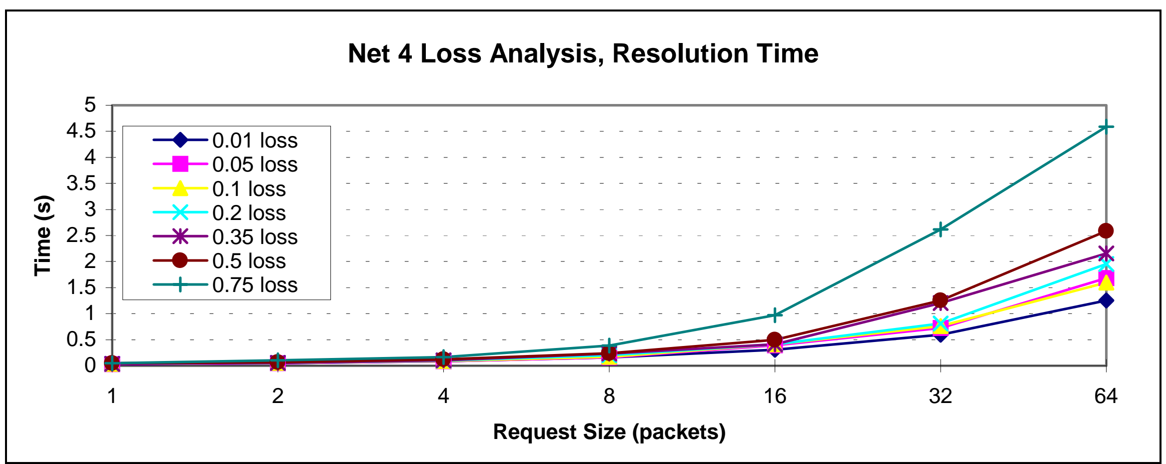

Figure 18. ALRP loss performance analysis for resolution time.

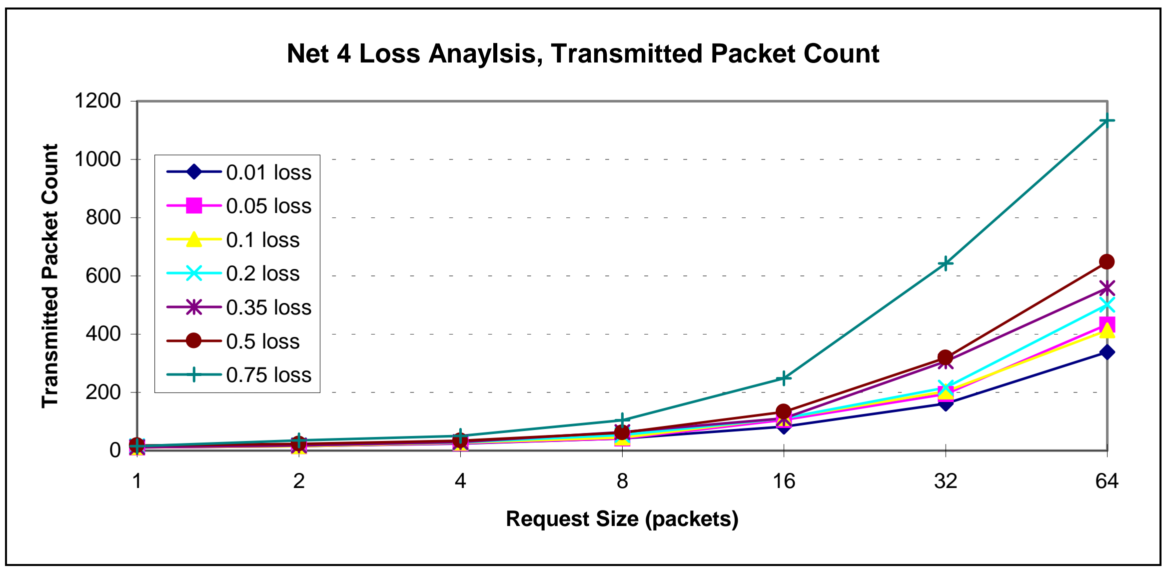

Figure 19. ALRP loss performance analysis for number of transmitted packets.

Regardless of the error rate, the ALRP service worked. Comparing the 75 percent packet loss case to the one percent packet loss case, the worst case packet count exhibited less than a four

times increase in the resolution time and a 3.3 times increase in the number of packets. The performance difference and network impact at low packet counts was insignificant. For the expected request size of one to two packets, there was effectively no difference between different error rates. These results indicate that ALRP scales well under network loading with associated packet loss.

6.4.3 ALRP Scalability Analysis

The most important analysis is to determine if ALRP is scalable. That is, to determine if ALRP performance increases at a less than exponential rate for a unit increase in the TTL count. TTL counts, for the continuous design, have a direct relationship to ALRP performance. For each increase in the TTL for an unloaded network, the delay for one packet increases. The number of receivers does not matter as they are only passive participants in the protocol. Only a server that has the proper library responds to the request. The cases presented in Table 7 increase the TTL values by different amounts. Thus, to achieve an accurate perspective on the scalability, the resolution time must be plotted against the TTL values. In this case, the plot will be a function of the link count, which is one less than the TTL value. Additionally, case 1 and case 2 have the same TTL value. Since case 2 is the case that is more uniform to the other cases, case 1 is discarded.

Figure 20 and Figure 21 show the scalability analysis for the one percent and five percent error loss. These plots give the resolution time as a function of the number of links and the transmitted packet count as a function of the number of links, respectively. The results presented are an average of the normalized data set. Each data set was normalized to the case where the request size was one packet. The average of the four cases are plotted. For each loss rate, a linear "best-fit" curve is generated and its equation and R^2 value are presented. The closer the R^2 value is to unity, the more accurate the linear regression.

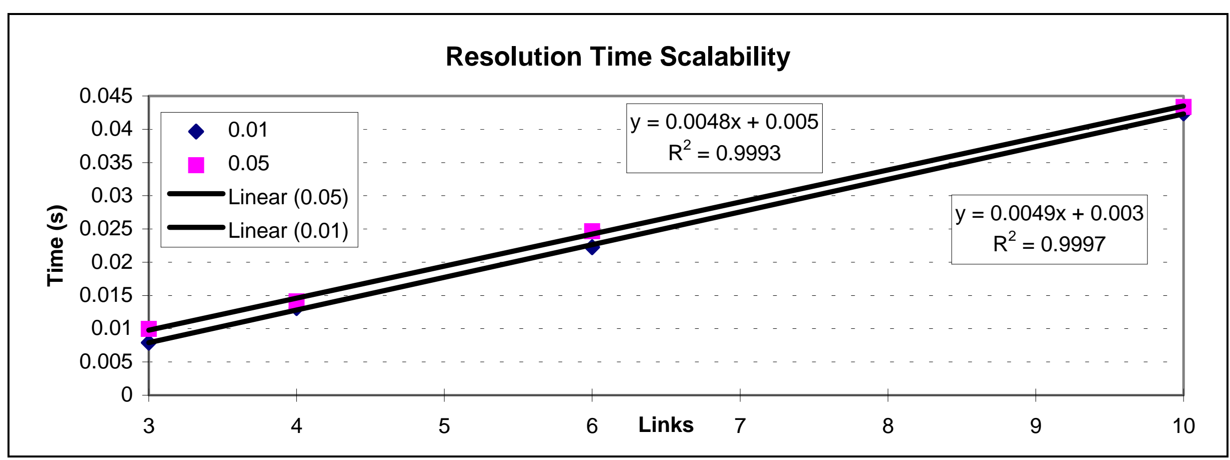

Figure 20. ALRP resolution time scalability.

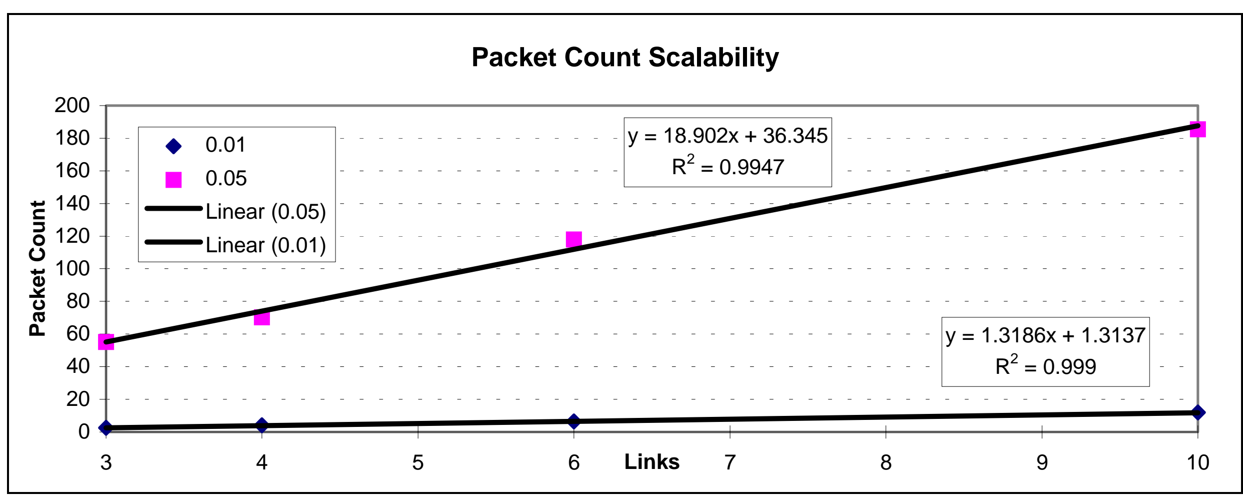

Figure 21. ALRP packet count scalability.

Figure 20 shows a linear correlation for both the one percent and five percent resolution time cases. The slope for both curves is significantly less than one. Also, both equations are almost the same. This indicates that resolution time scales well, regardless of error rates, and that results for larger networks can be determined by the two similar equations. For transmitted packet counts of Figure 21, the one percent error case has a slope of 1.3 and the five percent error case has a slope of 19. This indicates that low loss rates scale significantly better than higher loss rates with respect to the transmitted packet counts. Thus, the impact on the network is insignificant for the one percent case and moderate for the five percent case for moderately sized to large networks. From the above analysis, indications are that unrealistically huge networks will have a large number of retransmitted packets at five percent loss. Source to destination distances in modern internets generally are not much more than 15 hops. The significant difference in the two transmitted packet count equations indicates that higher error rates will lead to significantly more packets being retransmitted. Excessive numbers of packets will begin to degrade resolution time performance as network capacity is exceeded.

Caching is an important feature of a system that affects scalability by moving data closer to the requesting node. For example, if the average distance between the source and the correct library server was five hops, caching may improve that to three hops. The analysis performed in this section estimates performance based on the network diameter, which provides an indication of performance regardless of the average distance. Without a deployed system, there is no way to know the actual average distance.

Proper analysis of multiple ALRP sessions requires an understanding of the distribution of servers, the request rates, the request sizes, the request distribution, and caches. Previous analysis indicates that request sizes, which is an important factor for multiple simultaneous sessions, are typically only a few packets. The other statistics are not known and an educated guess is only slightly better than a random guess, so no formal analysis was performed. Considering that the expected operating mode of a few packets has negligible impact on the network, a reasonable distribution of request rates and servers should have an insignificant

impact on network loading. At one percent loss, several hundred requests per second can be handled before the network is fully loaded. It is highly unlikely that individual nodes will make more than a few requests per day and that these requests will be of global scope every time. If all the requests are of global scope, this means that the caching strategy has failed. The more likely scenario is that the network will only see a request every few minutes, on average. And as most people at a site use the same software, caching should isolate these requests to local networks.

6.5 Summary

This chapter presented measurement results from both the prototype and simulation systems. The five different criteria and metrics of policy constraints, extensibility, overhead, resolution time, and scalability were applied to ATP and ALRP. ATP and ALRP are clearly extensible and can easily satisfy reasonable policy constraints. ATP, which is clearly not the ideal design for an active code transport protocol, has overhead and resolution times comparable to HTTP. ATP is used to demonstrate the requirements of an active code transport protocol and to provide functionality for the active library resolution service. Other researchers are developing active code transport protocols [44, 88]. ALRP overhead is clearly acceptable when compared to the typical overhead specified by IP. Analysis of resolution time and scalability of ALRP was performed through the use of the simulation system which was validated for one case by the prototype. The analysis results indicate that ALRP has low resolution times and should be scalable for typical error rates and network sizes.

Chapter 7. Conclusions

Active networks constitute a new area of research and have the potential to revolutionize the way computer networks are used and implemented. The paradigm shifts from standardizing how bits are ordered in the network to standardizing how computation is performed on the bits in the network. This new model uses a higher level of abstraction to allow rapid and automated deployment of new network technologies. To support the active network paradigm, a mechanism must be developed to automatically locate and retrieve required active code from the network. The resolution of active code is the focus of this work. This chapter summarizes the dissertation, discusses objectives and contributions, and outlines future areas of research.

7.1 Objectives

The first objective of this research was to investigate potential strategies for the resolution of active libraries in an active network. This investigation included determining operating system environment and network environment requirements, which are presented in Chapter 3. The second objective of this research was to propose a specific strategy to solve the problem of active library resolution, which is discussed in Chapter 4. The third objective of this research was to develop metrics to measure the "goodness" of a resolution approach. The resulting metrics are resolution time, overhead, and scalability and the resulting criteria are extensibility and support of policy constraints. Evaluation of the proposed strategy requires development of a proof-of-concept system, measurement methods, and performance experiments. The fourth and last objective of this research was to implement selected portions of the proposed system to evaluate the resolution strategy against the specified metrics. This evaluation was performed through the use of a prototype system and a simulation system. The implementation is discussed in Chapter 5 and the evaluation is presented in Chapter 6.

7.2 Approach

The research approach was to determine requirements for the supporting environment and its interaction with the proposed resolution service, as documented in Chapter 3. Once the requirements and concepts were outlined, the proposed resolution approach of Chapter 4 was investigated. This investigation included determination of the appropriate demonstration and evaluation mechanisms. The environmental assumptions, proposed approach, and demonstration and evaluation methodology are presented in the dissertation. The implementation strategy was to determine and remove features that have no bearing on the evaluation of the final system. A prototype implementation was created to verify the assumptions and help refine the specifications, as discussed in Chapter 5. The final prototype and simulation systems were evaluated and the results proving the validity and goodness of the approach are presented in Chapter 6.

This minimal prototype delivers an IPv4 ping application from one node to another node. This pre-compiled ping application was forwarded, not copied, across the network and required a

library that was resolved from the network. Measurements on how this demonstration system worked and supporting measurements from the simulation system were used to justify that the solution approach is a “good” one.

7.3 Results

The evaluation of the design, as presented in Chapter 6, was based on five metrics and design criteria, overhead, resolution time, scalability, extensibility, and support of policy constraints. Extensibility and support of policy constraints are inherent in the system. A large percentage of the overhead is reduced by the introduction of compressed MIME headers. The precise overhead is hard to estimate but an approximate measurement was made. Because of the extensibility requirements, a certain amount of overhead must be accepted. The ideal time for resolution times was specified to be 0.3 seconds to ten seconds and the simulation results show resolution times of 0.1 seconds to 4.6 seconds, for varying network diameters and network loads. For the expected operating mode of a few network packets, 0.1 seconds is an extremely good resolution time with negligible impact on network resources. The final metric was scalability. The analysis indicated that the resolution time was clearly linear for increasing network diameters and, thus, the strategy is scalable.

7.4 Contributions

The proposed research makes a number of contributions to active networks and related fields. The primary contribution is the development of a comprehensive approach for active library resolution in active networks. This approach can be directly applied to support mobile software agents and extensible operating system libraries. Related areas that can also use a similar system include distributed and standard application programming as well as Web “servlets.” One can also argue that concepts from the resolution strategy could be applied to FTP archives.

The resolution strategy for active libraries is based on simple and extensible transport headers that are used in a transport protocol for active code. These headers use standard MIME constructs and can represent any characteristic of the library. Like FTP, each component of a distributed registry has individual administration that allows for rapid growth of the system. The distributed registry can be searched and constrained, in numerous ways, for the headers that describe an active library. To increase scalability and reduce resolution times, downloaded libraries can be cached for future requests.

This research takes a slightly different view of the extensible operating system from the existing literature [7-18] in that the basic resource protection features of the operating system, a network protocol that includes ATP, and a library resolution strategy (ALRP) are the only components required for an active network operating system. Using the portability concepts behind active networks, every other part of the operating system can be retrieved from the network, on demand. This unique perspective allows the user to view the operating system as the same across any hardware platform. Active code becomes more than specialized network protocols; it becomes the computing paradigm on all levels.

The research also provides a potentially different definition of some required features of the operating environment than existing systems [2-18]. These are the basic resource access primitives, how the operating system handlers are defined, and a view of how the operating system itself could be implemented. Though not implemented, the use of dynamic compilation and code safety mechanisms instead of interpreters is preferred.

One other contribution is a more comprehensive method to transport active code across the network than that provided by existing research [1-7, 44, 88]. This transport mechanism is defined in the active transport protocol (ATP). Three new key features of the transport mechanism are usage payment, security, and means to verify that the active code was successfully delivered and executed. ATP was also designed to allow processing by intermediate nodes and, with minor modifications, to be easily adapted to different internet protocols and also to serve as a simple internet protocol. In the design of the transport protocol, a scheme for compressed MIME headers was specified and used. Compression of protocol headers can lead to a substantial reduction in overhead which becomes important for search scalability. A TCP-based scheme for active code transport, as used in ATP, is clearly not the ideal method — an IP-based method is better. However, ATP is sufficient to show what is required of active code transport and how it could be implemented. Recent investigations into IP-based methods can be found in [44, 88].

The final contribution is the active library resolution protocol (ALRP), which provides the actual mechanism to perform the query for active libraries. This protocol uses expanding-ring multicast searches, relies on compressed MIME headers, and allows intermediate processing nodes to process ALRP data. ALRP is designed to allow queries on MIME headers, which allow any type of properly defined policy constraints to be imposed on the search. A related result is the definition of primitive interface specifications for active library resolution. The design of ALRP also results in a limitation of approximately 590 different functions in any given library.

7.5 Future Research

Active networks are a new area of research and will be a research area for some time to come. The concepts of an active network make fundamental changes to how the network is viewed and used. This work focuses on strategies for active library resolution and did not directly investigate many related issues. Possible future areas of research include 1) investigation of how the proposed resolution architecture can fit into an operating system, 2) determination of how the minimal function set of an extensible operating system, the base protection mechanisms, the resolver, and a network protocol, would work, 3) research into how the system could be applied to other architectures such as the World-Wide Web and electronic mail, 4) development of improved active code transport protocols, 5) creation of compression schemes for protocol interfaces, 6) implementation of ASN.1 interface conversion, 6) investigation of cache and translation servers, and 7) development of precise active library commerce and distribution models.

The resolution of programming libraries from the network is also an interesting concept for standard applications. Many dynamically linked X Windows programs often result in error messages due to various shared libraries not being found or being somehow incorrect. The typical method to fix the problem is to have an administrator install the correct libraries. However, manual human intervention absolutely does not work for an active network. This is where the strategy for active library resolution comes into its own.

References

[1] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden, "A Survey of Active Network Research," *IEEE Communications Magazine*, Vol. 35, No. 1, January 1997, pp. 80-86. Available at http://www.tns.lcs.mit.edu/publications/ieeecomms97.html

[2] D.L. Tennenhouse and D.J. Wetherall, "Towards an Active Network Architecture," *Computer Communication Review*, Vol. 26, No. 2, April 1996, pp. 5-18. Available at http://www.tns.lcs.mit.edu/publications/ccr96.html

[3] Y. Yemini and S. da Silva, "Towards Programmable Networks," *Proceedings FIP/IEEE International Workshop on Distributed Systems*, October 1996 (26 May 1997), http://www.cs.columbia.edu/~dasilva/content/netscript/pubs/dsom96.ps

[4] S. Bhattacharjee, K.L. Calvert, and E.W. Zegura, "On Active Networking and Congestion," *Technical Report,* GIT-CC-96/02*,* Georgia Institute of Technology, 1996. Available at http://www.cc.gatech.edu/fac/Ellen.Zegura/anandc.ps

[5] J. Hartman, U. Manber, L. Peterson, and T. Proebsting, "Liquid Software: A New Paradigm for Network Systems," *Technical Report,* TR 96-11, University of Arizona, June 1996. Available at ftp://ftp.cs.arizona.edu/xkernel/Papers/tr96-11.ps

[6] J.M. Smith, D.J. Farber, C.A. Gunter, S.M. Nettles, D.C. Feldmeier, and W.D. Sincoskie, "SwitchWare: Accelerating Network Evolution," *White Paper*, 26 June 1996. Available at http://www.cis.upenn.edu/~jms/white-paper.ps

[7] J.S. Shapiro, S.J. Muir, J.M. Smith, and D.J. Farber, "Operating System Support for Active Networks," *Unpublished Paper*. Available at http://www.cis.upenn.edu/~eros/devel/sigcomm97.300dpi.ps

[8] D.J. Wetherall, J.V. Guttag, and D.L. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," *Submitted Paper*, *IEEE OPENARCH 1998*. Available at http://www.tns.lcs.mit.edu/publications/openarch98.html

[9] U. Legedza, D.J. Wetherall, and J. Guttag, "Improving the Performance of Distributed Applications Using Active Networks," Submitted Paper, *IEEE INFOCOM 1998*. Available at http://www.sds.lcs.mit.edu/publications/infocom98lwg.html

[10] D.M. Murphy, "Building an Active Node on the Internet," *M.S. Thesis*, Massachusetts Institute of Technology, May 1997. Available at http://www.sds.lcs.mit.edu/publications/murphy97.html

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," *RFC 2068*, January 1997.

[12] D.R. Engler, M.F. Kaashoek, and J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Operating Systems Review*, Vol. 29, No. 5, December 1995, pp. 251-266. Available at http://www.pdos.lcs.mit.edu/~engler/sosp-95.ps

[13] D.R. Engler and M.F. Kaashoek, "Exterminate All Operating System Abstractions," *Proceedings Fifth Workshop on Hot Topics in Operating Systems (HotOS-V),* Orcas Island, Washington, 4-5 May 1995, pp. 78-83. Available at http://www.pdos.lcs.mit.edu/~engler/hotos-jeremiad.ps

[14] D.R. Engler, M.F. Kaashoek, and J. O'Toole, "The Operating System Kernel as a Secure Programmable Machine," *Operating Systems Review*, Vol. 29, No. 1, January 1995, pp. 78-82. Available at http://www.pdos.lcs.mit.edu/~engler/xsigops.ps
[15] D.A. Wallach, D.R. Engler, and M.F. Kaashoek, "ASHs: Application-specific Handlers for High-performance Messaging," *Computer Communication Review*, Vol. 26, No. 4, October 1996, pp. 40-52. Available at http://www.pdos.lcs.mit.edu/~engler/sigcomm96.ps
[16] A.B. Montz, D. Mosberger, S.W. O'Malley, L.L. Peterson, T.A. Proebsting, and J.H. Hartman, "Scout: A Communications-Oriented Operating System," *Technical Report*, TR 94-20, Department of Computer Science, University of Arizona, 17 June 1994. Available at ftp://ftp.cs.arizona.edu/xkernel/Papers/scout.ps
[17] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chamers, and S. Eggers, "Extensibility, Safety, and Performance in the SPIN Operating System," *Operating Systems Review*, Vol. 29, No. 5, December 1995, pp. 267-84. Available at http://www.cs.washington.edu/research/projects/spin/www/papers/SOSP95/sosp95.ps
[18] S. Savage and B.N. Bershad, "Issues in the Design of an Extensible Operating System," *Unpublished Paper*, 28 June 1994 (17 May 1997), http://www.cs.washington.edu/research/projects/spin/www/docs/issues_ext.ps
[19] M.E. Fiuczynski and B.N. Bershad, "An Extensible Protocol Architecture for Application-Specific Networking," *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, California, 22-26 January 1996, pp. 355-64. Available at http://www.cs.washington.edu/homes/mef/usenix96.html
[20] J. Gosling, B. Joy, and G. Steele, *The Java Language Specifications, Version 1.0*, Addison-Wesley, August 1996. Available at http://www.javasoft.com:80/docs/language_specification.html
[21] N.S. Borenstein, "EMail With a Mind of Its Own: The Safe-TCL Language for Enabled Mail," *IFIP Transactions C, Communications Systems*, Barcelona, Spain, 1-3 June 1994, pp. 389-402. Available at http://www.cs.utah.edu/~ampsem/ulpaa-94.txt
[22] "Arizona Computer Science: Sumatra Project," *Web Document,* 25 February 1997 (7 May 1997), http://www.cs.arizona.edu/sumatra/
[23] D.R. Engler and M.F. Kaashoek, "DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation," *Computer Communication Review*, Vol. 26, No. 4, October 1996, pp. 53-59. Available at http://www.pdos.lcs.mit.edu/~engler/dpf-sigcomm96.ps
[24] D.R. Engler and T.A. Proebating, "DCG: An Efficient, Retargetable Dynamic Code Generation System," *SIGPLAN Notices*, Vol. 29, No. 11, November 1994, pp. 263-272. Available at http://www.pdos.lcs.mit.edu/~engler/dcg.ps
[25] D. Keppel, S.J. Eggers, and R.R. Henry, "A Case for Runtime Code Generation," *Technical Report*, UW-CSE-91-11-04, Department of Computer Science and Engineering, University of Washington, November 1991. Available at ftp://ftp.cs.washington.edu/tr/1991/11/UW-CSE-91-11-04.PS.Z
[26] D.R. Engler, "VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System," *SIGPLAN Notices*, Vol. 31, No. 5, May 1996, pp. 160-170. Available at http://www.pdos.lcs.mit.edu/~engler/vcode-pldi.ps
[27] M. Poletto, D.R. Engler, and M. F. Kaashoek, "tcc: A Template-Based Compiler for 'C," *ACM SIGPLAN Conference on Programming Language Design and Implementation*

(PLDI) 1997, *To Appear*. Available at http://www.pdos.lcs.mit.edu/~engler/tickc-pldi-submit.ps

[28] M. Poletto, D.R. Engler, and M. F. Kaashoek, "tcc: A System for Fast, Flexible, and High-Level Dynamic Code Generation," *First Annual Workshop on Compiler Support for System Software,* 23-24 February 1996 (26 May 1997), http://www.pdos.lcs.mit.edu/~engler/wcsss.ps

[29] D.R. Engler, W.C. Hsieh, and M.F. Kaashoek, "'C: A Language for High-level, Efficient, and Machine-independent Dynamic Code Generation," *Proceedings of 23rd Annual ACM SIGACT-SIGPLAN Symposium*, St. Petersburg Beach, Florida, 21-24 January 1996, pp.131-144. Available at http://www.pdos.lcs.mit.edu/~engler/pldi-tickc.ps

[30] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad, "Fast, Effective Dynamic Compilation," *SIGPLAN Notices,* Vol. 31, No. 5, May 1996, pp. 149-159. Available at http://www.cs.washington.edu/research/projects/unisw/DynComp/www/Papers/pldi96.ps.Z

[31] D. Keppel, S.J. Eggers, and R.R. Henry, "Evaluating Runtime-Compiled Value-Specific Optimizations," *Technical Report,* UW-CSE-93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993. Available at ftp://ftp.cs.washington.edu/tr/1993/11/UW-CSE-93-11-02.PS.Z

[32] B.N. Bershad, S. Savage, P. Pardyak, D. Becker, M. Fiuczynski, and E.G. Sirer, "Protection is a Software Issue," *Proceedings 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, Washington, 4-5 May 1995, pp. 62-65. Available at http://www.cs.washington.edu/research/projects/spin/www/papers/HotOS95/hotos95.ps

[33] P. Deutsch and C.A. Grant, "A Flexible Measurement Tool for Software Systems," *Information Processing 71*, 1971, pp. 320-326.

[34] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham, "Efficient software-based fault isolation," *Operating Systems Review*, Vol. 27, No. 5, December 1993, pp. 203-216.

[35] M. Franz, "Dynamic Linking of Software Components," *IEEE Computer*, Vol. 30, No. 3, March 1997, pp 74-81.

[36] P. Pardyak and B.N. Bershad, "Dynamic Binding for an Extensible System," *Operating Systems Review*, Vol. 30, Special Issue, October 1996, pp. 201-212. Available at http://www.cs.washington.edu/research/projects/spin/www/papers/OSDI96/eventsOSDI96.ps

[37] E.G. Sirer, M.E. Fiucynski, P. Pardyak, and B.N. Bershad, "Safe Dynamic Linking in an Extensible Operating System," *First Annual Workshop on Compiler Support for System Software*, 23-24 February 1996 (26 May 1997), http://www.cs.washington.edu/research/projects/spin/www/papers/WCS/domain.ps

[38] P. Pardyak, S. Savage, and B.N. Bershad, "Language and Runtime Support for Dynamic Interposition of System code," *Unpublished Paper*, 3 November 1995 (17 May 1997), http://www.cs.washington.edu/research/projects/spin/www/papers/WCS/dispatcher.ps

[39] D. Volpano, "Provably-Secure Programming Languages for Remote Evaluation," *SIGPLAN Notices*, Vol. 32, No. 1, January 1997, pp. 117-119.

[40] R. Crelier, "Extending Module Interfaces without Invalidating Clients," *Software-Concepts and Tools*, Vol. 16, No. 2, 1995, pp. 49-62.

[41] S.M. Dorward, R. Sethi, and J.E. Shopiro, "Adding New Code to a Running C++ Program," *USENIX C++ Conference*, San Francisco, California, 9-11 April 1990, pp. 279-292.
[42] M. Shapiro, "Binding Should be Flexible in a Distributed System," *Proceedings Third International Workshop on Object Orientation in Operating Systems*, Asheville, North Carolina, 9-10 December 1993, pp. 216-217.
[43] V.C. Van, "A Defense Against Address Spoofing Using Active Networks," *M.S. Thesis*, Massachusetts Institute of Technology, May 1997. Available at http://www.sds.lcs.mit.edu/publications/van97.html
[44] D.J. Wetherall and D.L. Tennenhouse, "The ACTIVE IP Option," *Proc. 7th ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996 (26 May 1997), http://www.tns.lcs.mit.edu/publications/sigops96ws.html
[45] D.S. Alexander, M. Shaw, S.M. Nettles, and J.M Smith, "Active Bridging," *Computer Communication Review*, Vol. 27, No. 4, October 1997, 101-111.
[46] "GNU C Library - GNU Project - Free Software Foundation (FSF)," *Web Document*, 16 May 1997 (17 May 1997), http://www.gnu.org/software/libc/libc.html
[47] O. Tawakol and N. Singh, "A Name Space Context Graph for Multi-Context Systems," *Web Document*, March 1995 (17 May 1997), http://logic.stanford.edu/sharing/papers/ontology.ps
[48] N. Leser, "The Distributed Computing Environment Naming Architecture," *Distributed Systems Engineering*, Vol. 1, No. 1, September 1993, pp. 19-28.
[49] K.R. Sollins and D.D. Clark, "Distributed Name Management," *Proceedings of the IFIP TC 6/WG 6.5 Working Conference*, Munich, West Germany, 27-29 April 1987, pp. 97-115.
[50] S. Benford and O.-K. Lee, "Collaborative Naming in Distributed Systems," *Distributed Systems Engineering*, Vol. 1, No. 2, December 1993, pp. 67-74.
[51] S. Benford and O.-K. Lee, "A Naming Model for Distributed Group Communication," *IFIP Transactions C, Communication Systems*, Vol. C-25, 1-3 June 1994, pp. 357-370.
[52] "IBM Aglets Workbench - Home Page," *Web Document*, 4 April 1997 (17 May 1997), http://www.trl.ibm.co.jp/aglets/
[53] T. Finn, "UMBC Agent Web," *Web Document*, (17 May 1997), http://www.cs.umbc.edu/agents/
[54] M. Wooldridge and N.R. Jennings, "Intelligent Agents: Theory and Practice," *Knowledge Engineering Review*, Vol. 10, No. 2, June 1995, pp. 115-152. Available at http://www.elec.qmw.ac.uk/dai/
[55] H.S. Nwana, "Software Agents: An Overview," *Knowledge Engineering Review*, Vol. 11, No. 3, October/November 1996, pp. 205-244. Available at http://www.cs.umbc.edu/agents/introduction/ao/
[56] A. Lingnau and O. Drobnik, "An Infrastructure for Mobile Agents: Requirements and Architecture," *Web Document*, 1995 (21 May 1997), ftp://ftp.tm.informatik.uni-frankfurt.de/pub/papers/agents/13dis-paper.ps.gz
[57] C.G. Harrison, D.M. Chess, and A. Kershenaum, "Mobile Agents: Are They a Good Idea?," *Technical Report*, IBM Research Division, 28 March 1995. Available at http://www.research.ibm.com/xw-d953-mobag-ps

[58] I. Ben-Shaul and S. Ifergan, "WebRule: An Event-based Framework for Active Collaboration Among Web Servers," *Hyper-proceedings of the Sixth International World-Wide Web Conference*, Santa Clara, California, 7-11 April 1997, http://www6.nttlabs.com/papers/PAPER45/PAPER45.html

[59] M.R. Genesereth and N.P. Singh, "A Knowledge Sharing Approach to Software Interoperation," *Web Document*, 31 January 1994 (17 May 1997), http://logic.stanford.edu/sharing/papers/ksa.ps

[60] M.R. Genesereth, N.P. Singh, and M.A. Syed, "A Distributed and Anonymous Knowledge Sharing Approach to Software Interoperation," *Web Document*, 15 November 1994 (17 May 1997), http://logic.stanford.edu/sharing/papers/fgcs.ps

[61] T. Finn, J. Weber, G. Wiederhold, M. Genesereth, R. Fritzson, D. McKay, J. McGuire, R. Pelavin, S. Shapiro, and C. Beck, "Specification of the KQML Agent-Communication Language," *Web Document*, 15 July 1993 (17 May 1997), http://logic.stanford.edu/sharing/papers/kqml.ps

[62] Y. Labrou and T. Finin, "A Proposal for a new KQML Specification," *Technical Report,* TR-CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, February 1997. Available at http://www.cs.umbc.edu/~jklabrou/publications/tr9703.ps

[63] M.R. Genesereth, "Knowledge Interchange Format," *Web Document*, 13 September 1995 (17 May 1997), http://logic.stanford.edu/kif/specification.html

[64] P.F. Patel-Schneider and B. Swartout, "Description Logic Specification from the KRSS Effort," *Working Draft*, 30 June 1993. Available at http://www-ksl.stanford.edu/knowledge-sharing/papers/dl-spec.ps

[65] A. Quintero, M.E. Ucres, and S. Takahashi, "Multi-Agent System Protocol Language Specification," *Web Document*, (17 May 1997), http://www.cs.umbc.edu/~cikm/iia/submitted/voewomg/yubarta.html

[66] G.A.W. Vreeswijk, "Open Protocol in Multi-Agent Systems," *Technical Report,* Department of Computer Science, University of Limburg, January 1995. Available at http://www.cs.rulimburg.nl/~vreeswyk/opinmas.htm

[67] R. Srinivasan, "RPC: Remote Procedure Call Protocol Specification Version 2," *RFC 1831*, August 1995.

[68] R. Srinivasan, "XDR: External Data Representation Standard," *RFC 1832*, August 1995.

[69] R. Srinivasan, "Binding Protocols for ONC RPC Version 2," *RFC 1833*, August 1995.

[70] N. Brown and C. Kindel, "Distributed Component Object Model Protocol -- DCOM/1.0," *Internet Draft*. Work in progress.

[71] "The Common Object Request broker: Architecture and Specification: Revision 2.0," Open Management Group, July 1995. Available at http://www.omg.org/corba/corbiiop.htm

[72] SunSoft, "Java Remote Method Invocation Specification," Sun Microsystems, 10 February 1997. Available at http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html

[73] "Distributed Computing Environment," *Web Document*, (17 May 1997), http://www.opengroup.org/tech/dce/

[74] T. Berners-Lee, L. Masinter, and M. McCahill, "Uniform Resource Locators (URL)," *RFC 1738*, 20 December 1994.

[75] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan, "Service Location Protocol," *RFC 2165,* June 1997.
[76] P. Mockapetris, "Domain Names - Concepts and Facilities," *RFC 1034*, November 1987.
[77] P. Mockapetris, "Domain Names - Implementation and Specification," *RFC 1035*, November 1987.
[78] A. Gulbrandsen and P. Vixie, "A DNS RR for Specifying the Location of Services (DNS SRV)," *RFC 2052*, October 1996.
[79] R. Atkinson, "Security Architecture for the Internet Protocol," *RFC 1825*, August 1995.
[80] G. Malkin, and R. Minnear, "RIPng for IPv6," *RFC 2080*, 10 January 1997.
[81] J. Moy, "OSPF Version 2," *RFC 1583*, 23 March 1994.
[82] SunSoft, "NIS+ and FNS Administration Guide," Sun Microsystems, 1995.
[83] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound, "Dynamic Updates in the Domain Name System (DNS UPDATE) ," *RFC 2136*, April 1997.
[84] D.C. Lee and M. Abrams, *Ed.*, "Response Time Improvement," *WWW: Beyond The Basics*, Prentice-Hall, *To Appear*.
[85] D.C. Lee, "Pre-Fetch Document Caching to Improve World-Wide Web User Response Time," *M.S. Thesis*, Virginia Polytechnic Institute and State University, March 1996.
[86] SunSoft, "dlopen (3x) Miscellaneous Library Function," *Man Page*, Solaris 2.5, 21 February 1995.
[87] S. Thompson and T. Nartin, "IPv6 Stateless Address Autoconfiguration," *RFC 1971*, August 1996.
[88] D.S. Alexander, B. Braden, C.A. Gunter, A.W. Jackson, A.D. Keromytis, G.J. Minden, and D. Wetherall, "Active Network Encapsulation Protocol (ANEP)," *Internet Draft*, July 1997. Work in progress. Available at: http://www.cis.upenn.edu/~switchware/ANEP
[89] N. Borenstein and N. Freed, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," *RFC 2045*, 2 December 1996.
[90] D. Crocker, "Standard for the format of ARPA Internet text messages," *RFC 822*, STD 11, 13 August 1982.
[91] International Organization for Standardization, "Information Technology — Abstract Syntax Notation One (ASN.1): Specification of Basic Notation," *ISO/IEC 8824-1:1995*, 1 September 1996.
[92] S.E. Deering, "Multicast Routing in a Datagram Internetwork," *Ph.D. Thesis*, Stanford University, December 1991.
[93] P.B. Danzig, "Flow Control for Limited Buffer Multicast," *IEEE Transactions on Software Engineering,* Vol. 20, No. 1, pp. 1-12, January 1994.
[94] S.Y. Cheung and M.H. Ammar, "Using Destination Set Grouping to Improve the Performance of Window-Controlled Multipoint Connections," *Technical Report, GIT-CC-94-32*, Georgia Institute of Technology, August 1994.
[95] M.A. Jolfaei and U. Quernheim, "A New Selective Repeat ARQ Scheme for Multicast Communication," *IFIP Broadband Communications, II (C-24),* Paris, France, pp. 119-136, 2-4 March 1994.

[96] S. Pingali, D. Towsley, and J.F. Kurose, “A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols,” *Performance Evaluation Review*, Vol. 22, No. 1, pp. 221-230, May 1994.

[97] R. Yavatkar and L. Manoj, “Optimistic Strategies for Large-Scale Dissemination of Multimedia Information,” *Proceedings of ACM Multimedia 1993*, Anaheim, California, pp. 13-20, 1-6 August 1993.

[98] X. Jia, “A Total Ordering Multicast Protocol Using Propagation Trees,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 6, pp. 617-627, June 1995.

[99] R. AIello, E. Pagani, and G.P. Rossi, “Design of a Reliable Multicast Protocol,” *IEEE INFOCOM 1993*, San Francisco, California, pp. 75-81, 30 March-1 April 1993.

[100] Droms, R., “Dynamic Host Configuration Protocol,” *RFC 1531*, October 1993.

[101] R. Braden, “Requirements for Internet Hosts -- Application and Support,” *RFC 1123*, October 1989.

[102] D. Crocker and P. Overell, “Augmented BNF for Syntax Specifications: ABNF,” *RFC 2234*, November 1997.

[103] M.Liljeberg, H. Helin, M. Kojo, and K. Raatikainen, “Mowgli WWW Software: Improved Usability of WWW in Mobile WAN Environments,” *IEEE GLOBECOM 1996,* London, United Kingdom, 18-22 November 1996, pp. 33-37.

[104] N. Freed, J. Klensin, and J. Postel, “Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures,” *RFC 2048*, 28 January 1997.

[105] “Coded Character Set - 7-Bit American Standard Code for Information Interchange,” *ANSI X3.4-1986*, 1986.

[106] A. Emtage and P. Deutsch, “archie — An Electronic Directory Service for the Internet,” *Proceedings of the Winter 1992 USENIX Conference*, San Francisco, California, 20-24 January 1992, pp. 93-110.

[107] A. Luotonen and K. Altis, “World-Wide Web Proxies,” *Proceedings of the First International World-Wide Web Conference*, 1994 (August 1995) http://www1.cern.ch/PapersWWW94/luotonen.ps

[108] Sun Microsystems, Inc, “regcomp - C Library Function,” *MAN Page*, Solaris 2.5, 26 January 1995.

[109] The Regents of the University of California, *ping*, 15 December 1993.

[110] A. Ballardie, “Core Based Trees (CBT) Multicast Routing Architecture,” *RFC 2201,* September 1997.

[111] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei, “Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification,” *Internet Draft*, 9 September 1997. Work in progress.

[112] M.F. Kaashoek, T. Pinckney, and J.A. Tauber, “Dynamic Documents: Extensibility and Adaptability in the WWW,” *Second International WWW Conference '94: Mosaic and the Web*, October 1994 (23 August 1997), http://www.ncsa.uiuc.edu/SDG/IT94/IT94Info.html

[113] M.T. Ozsu, A. Munoz, and D. Szafron, “An Extensible Query Optimizer for an Objectbase Management System,” *Proceedings of the 1995 ACM CIKM International Conference on Information and Knowledge Management*, Baltimore, Maryland, 28 November-2 December 1995, pp. 188-196.

[114] R.P. Cook, "The Type Extensible Architecture of a Simple Database System," *Software Concepts and Tools*, 1996, Vol. 17, pp. 141-147.
[115] M. Auslander, H. Franke, B. Gamsa, O. Krieger, and M. Stumm, "Customization Lite," *Sixth Workshop on Hot Topics in Operating Systems*, Cape Cod, Maryland, 5-6 May 1997, pp. 43-48.
[116] B. Basdell, J. Favaro, and B. Bird, "An Open Software Architecture for a Scaleable and Extensible Software Engineering Environment," *Proceedings of the ESA 1996 Product Assurance Symposium and Software Product Assurance Workshop*, Noordwijk, Netherlands, March 1996, pp. 259-262.
[117] D.C. Schmidt, "A Family of Design Patterns for Application-Level Gateways," *Theory and Practice of Object Systems*, Vol. 2, No. 1, 1996, pp. 15-30.
[118] S. Wrobel, D. Wettschereck, E. Sommer, and W. Emde, "Extensibility in Data Mining Systems," *Proceedings of the 1996 International Conference on Knowledge Discovery and Data Mining*, Portland, Oregon, August 1996, pp. 214-219.
[119] Netscape Communications, Inc., *Fastrack Server,* Revision 2.01, 1996. Available at http://www.netscape.com
[120] L.J. Camp and M. Sibru, "Critical Issues in Internet Commerce," *IEEE Communications Magazine*, Vol. 35, No. 5, May 1997, pp. 58-62.
[121] N. Asokan, P.A. Janson, M. Steiner, and M. Waidner, "The State of the Art in Electronic Payment Systems," *IEEE Computer*, Vol. 30, No. 9, September 1997, pp. 28-35.
[122] Linux 2.1.43 Kernel Source, *FTP Site*, ftp://tsx-11.mit.edu/pub/linux/kernels/v2.1/linux-2.1.43.tar.gz
[123] "Multithreaded Routing Toolkit Home Page," *Web Document*, (22 August 1997) http://www.merit.edu/~mrt
[124] S Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," *Internet Draft*, July 1997. Work in progress.
[125] J. Postel, "Internet Protocol," *RFC 791*, September 1981.
[126] D.C. Lee and S.F. Midkiff, "A Sample Statistical Characterization of the World-Wide Web," *1997 IEEE Region 3 Southeast Conference*, Blacksburg, Virginia, 12-14 April 1997, pp. 174-178.
[127] "Welcome to the Linux Home Page," 19 June 1997 (24 June 1997), http://www.linux.org/
[128] "Programming Languages - C (ISO) (Revision and Redesignation of ANSI X3.159-1989); Amendment 1: C Integrity - 1995," *ANSI 9899*, 1995.
[129] "Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]," *IEEE 1003.1, 2nd Edition*, 12 July 1996.
[130] R. Gilligan, S. Thomson, J. Bound, and W. Stevens, "Basic Socket Interface Extensions for IPv6," *RFC 2133*, 21 April 1997.
[131] T. Narten, E. Nordmark, and W. Simpson, "Neighbor Discovery for IP Version 6 (IPv6)," *RFC 1970*, August 1996.
[132] S. Bradner, "Key Words For Use in RFCs to Indicate Requirement Levels," *RFC 2119*, March 1997.
[133] J. Postel, "Transmission Control Protocol," *RFC 793*, STD 7, 1 September 1981.

[134] Free Software Foundation, *regex,* Version 0.12, 1993. Available at ftp://prep.ai.mit.edu/pub/gnu/regex-0.12.tar.gz
[135] T.C. Bell, J.G. Cleary, and I.H. Witten, *Text Compression*, Englewood Cliffs: Prentice Hall, 1990.

Appendix A. Operating Environment Primitives

This appendix presents the set of general primitives that the operating environment and network services should provide. Material here is only a first attempt at the definition of primitives. A number of issues, primarily system security and safety, are not addressed by the primitives. Note that the primitives in Sections A.1 and A.2 were not implemented in the experimental system as they relate to the definition of the services provided by an active network operating system (ANOS). Most of the primitives in Section A.3 were implemented with some variations. As discussed in Chapter 3, this research does not implement an ANOS and the information in Sections A.1 and A.2 are presented only to provide the reader an understanding of some of the functionality required for an ANOS.

A.1 Environmental Services

There are two classes of services that the operating environment should provide. The first class includes services dealing with access to basic kernel functions. The second class provides uniform access to various operating system resources.

A.1.1 Kernel Services

This section presents standard kernel primitives that could be used to support an active network operating system. Security and safety issues are not addressed. These primitives are based on the Linux kernel calls [127] and the Sun dynamic linking calls [86]. All kernel primitives must be lightweight. It is possible to implement the system such that standard primitives, through the resolution service, can be obtained for a particular operating environment. However, this would defeat the purpose of a robust active network since developers must then implement the standard primitives for each environment.

`int kerror (void);`

This function returns the last error value. A list of error values can be found in the `kerror.h` header file. A return of ‘0’ indicates no error occurred. Each kernel function sets error status before the function returns. This is equivalent to a global error variable.

`void *kmalloc (int size);`

`kmalloc ()` allocates `size` bytes of kernel memory to the process. It returns a pointer to the allocated memory. If the pointer is `NULL`, then an error has occurred.

`void kfree (void* ptr);`

`kfree ()` returns to the system the memory allocated by `kmalloc ()`.

`int kgc (void);`

`kgc ()` forces a garbage collection on memory. Implementation of this function is optional. The function returns an integer value of `E_SUCCESS` or

E_NOT_IMPLEMENTED. In a multi-threaded environment, the return E_PROCESSING is defined and a supplemental routine, kgc_done (), is used to check to see if the garbage collection has been completed. Internal kernel signals may also be used to determine if the garbage collection has been completed. If multiple kgc () calls are made before the current call completes, they are ignored. Garbage collection is provided to support Java and other similar environments.

`void *klock (void *ptr, int size, int type);`

This function locks a region of memory as specified by ptr and size. It returns a handle to the locked memory or NULL if an error occurs. Two lock types are specified, write-only lock and read/write locks.

`int kunlock (void *handle);`

kunlock () unlocks a region of memory locked with klock (). If an invalid address is passed, an error is returned.

`int kaddtimer (struct timerinfo *timer);`

kaddtimer () is used to add a timer. The timerinfo structure contains the time of the event, the priority of the timer, general timer criteria, which system timer chain the timer should be added to, and what function should be executed when the specified timer criteria are met.

`int kdeltimer (struct timerinfo *timer);`

kdeltimer () deletes a timer added by kaddtimer (). If the timer is not found, then an error is returned. If a timer is being executed at the time of this call, the routine will not be able to delete the timer and returns E_TIMER_IN_PROGRESS. All timer operations are atomic.

`void *klib_open (long serial, int flag);`

klib_open () forces a dynamic load of an active library or active program as specified by serial. serial is a locally unique identifier assigned to active code. Note that library names are not used since there may be multiple libraries of the same name. The serial is obtained by examination of the symbol table. Returns a handle to the library or NULL if an error occurred.

`int klib_close (void *handle);`

klib_close () attempts to remove an active library or active program from kernel memory. If the library is in use, an error should be returned and the library is not removed. Process handling routines, which are not currently defined, must also be present.

`const void* klib_sym (void *handle, const char *symbol);`

klib_sym () is used to get a pointer to a function so that the function can be called. The function pointer is returned or NULL is returned if an error occurs. An active

network operating system may also need to return a parameter list. The parameter list is used to ensure that the function being called is the correct one.

`struct ksym_table *ksym_table_get (void);`

`ksym_table_get ()` returns a pointer to a copy of the entire symbol table, `struct ksym_table`. The symbol table contents have not be defined.

`int ksym_table_add (struct ksym *symbol);`

`ksym_table_add ()` is used to add a symbol, in `struct ksym`, to the system's symbol table. The calling function can specify symbol access privileges.

`int ksym_table_del (struct ksym *symbol);`

`ksym_table_del ()` deletes a symbol from the system's symbol table. Symbols in use, such as a program, may be not be deleted before their resources are released. An `E_SYM_IN_USE` error is returned if the symbol has related resources in use.

A.1.2 Resource Services

A number of resource services, which include IPC, threads, file systems, and so forth, must also be defined. The best definition is the standard ANSI C [128] and POSIX [129] functions for these services. Since there are several dozen of these, they will not be repeated here.

A.2 Network Services

Network primitives are a class of resource services that must be uniformly implemented by active networks. All these functions must be light-weight.

A.2.1 Device Interface Primitives

The following three primitives, which are used to translate interface names to an internal numeric representation, are from RFC 2133 [130]. The rest are independently defined.

`unsigned int if_nametoindex (const char *ifname);`

This routine translates an interface name, such as "le0" or "eth0," to an internal interface index. A negative return value indicates that the interface does not exist.

`char *if_indextoname (unsigned int ifindex, char *ifname);`

This routine translates an internal interface index to an interface name. If the translation fails, `NULL` is returned.

`struct if_nameindex *if_nameindex (void);`

This routine returns a list of all interfaces in the system. If no interfaces are in the system, `NULL` is returned.

`dev_struct *dev_open (unsigned int ifindex);`

dev_open () performs device initialization of an interface specified by ifindex. If the device failed to open, NULL is returned.

`int dev_close (dev_struct *dev);`

dev_close () deinitializes a device opened by dev_open (). The final status of the device is returned.

`dev_stat_struct *dev_stat (dev_struct *dev);`

dev_stat () returns status information about an open device. Status information includes current state, pending state, access and usage statistics, and configuration settings.

`int *dev_ctrl (dev_struct *dev, dev_stat_struct *stat);`

dev_ctrl () sets configuration settings for a open device. The configuration settings may be read by calling dev_stat ().

`int dev_select (DEV_SET *read, DEV_SET *write, DEV_SET *error, struct sys_time *timeout)`

dev_select () performs a blocking read and/or write on a open device. If the timeout is zero time units, the call will block forever. If the timeout is negative, the call will perform a poll. Asynchronous signal driven I/O should also be supported by the system.

`struct netbuff *dev_read (unsigned int ifindex, unsigned int protoindex, struct sys_time *timeout);`

This routine obtains a network buffer, stored in struct netbuff, from a network device identified by ifindex. The network buffer is for a specific network protocol, as identified by protoindex. Currently, the format for struct netbuff is not defined. The system will wait for timeout seconds if it is provided. A value of NULL for timeout indicates a non-blocking return whereas a value of zero for timeout indicates the system should wait until the request is completed.

`int dev_write (unsigned int ifindex, unsigned int protoindex, struct sys_time *timeout, struct netbuff *ptr);`

This routine writes data, stored in ptr, to a network interface, ifindex. The particular protocol to use is defined by protoindex. The system will try for timeout seconds if it is provided. A value of NULL for timeout indicates a non-blocking return whereas a value of zero for timeout indicates the system should wait until the request is completed.

`unsigned int proto_nametoindex (const char *protoname);`

This routine translates a protocol name to an internal protocol index. A negative return value indicates that the protocol does not exist.

`char *proto_indextoname (unsigned int protoindex)`
This routine translates an internal protocol index to a protocol name.

`void *proto_handler (unsigned int protoindex, char *symbol);`
This routine returns the function that provides service for the protocol as specified in `protoindex`. The service is the generic `dev_read`, `dev_write`, `dev_open`, `dev_close`, or similar function defined for the specific protocol.

`struct proto_nameindex *proto_nameindex (void);`
This routine obtains the system's list of installed protocols. If no protocols are installed, the system will return `NULL`.

`int proto_add (struct proto_index *prototype);`
This routine adds a protocol, specified in `prototype`, to the system. The `prototype` structure includes function pointers to default status, configuration, read, write, and other handlers. A negative return value indicates an error occurred.

`int proto_del (struct proto_index *prototype);`
This routine deletes a protocol from the system. A negative return value indicates an error occurred.

`struct net_fib *rtable_get (unsigned int protoindex);`
This routine obtains the route table, `struct net_fib`, for a specific protocol as denoted in `protoindex`. The system route table is not currently defined. If the protocol does not exist, `NULL` is returned.

`int rte_add (struct net_route *route);`
This routine adds a route to the system routing table. A negative return value indicates an error occurred.

`int rte_del (struct net_route *route);`
This routine deletes a route from the system routing table. A negative return value indicates an error occurred.

`struct ndcache *nd_cache_get (void)`
This routine obtains the system's physical address to network address translation cache. It is termed neighbor discovery, or "nd," rather than "arp" to remain consistent with IPv6 terminology [131]. A return value of `NULL` indicates a system error occurred.

`struct ndcache *nd_eui64tonet (unsigned char *eui64addr, int size);`
This routine obtains a network address, or set of addresses, for a given EUI-64 interface address. If the address is not in the cache, the routine will attempt to obtain the translation. If the return value is `NULL`, the translation does not exist. If more than one

address translation is present, the `next` pointer in `struct ndcache` will point to the next address.

`struct ndcache *nd_nettoeui64 (unsigned char *netaddr, int size);`

This routine obtains a physical, or EUI-64, address for a given network address. If the address is not in the cache, the routine will attempt to obtain the translation. If multiple hosts respond, a duplicate address error will be returned. If the translation does not exist, `NULL` will be returned.

`int nd_cache_add (struct ndcache *addr);`

This routine adds an address translation to the system cache. A negative return value indicates an error occurred.

`int nd_cache_del (struct ndcache *addr);`

This routine deletes an address translation from the system cache. A negative return value indicates an error occurred.

A.2.2 Socket Primitives

Active programs that implement a higher level protocol should also implement the standard POSIX socket calls [129]. These are not defined here.

A.3 Resolution Handler Primitives

The functions presented in this appendix describe the services available from an ATP and an ALRP compliant resolution handler. These functions are only used in an active network operating system. They are defined so that any given active code can call these routines without fail. They are also defined so that the basic underlying protocols can be changed without any change in the service primitive. The prototype implementation does not strictly meet the specifications given here as the prototype does not pass header values in the function calls; for rapid implementation, a separate call to an internal routine is used to transfer and receive headers. The prototype also supports extra functionality that is not currently used. Another difference is that the primitives below were specified for operation within an active network operating system. Some of the functions below assume the functional support of an active network operating system and those were not implemented. The generic specifications that are listed below assume that various internal active network operating system functions are present.

`int res_add_api (int ht, char *hname, struct modinfo *mod);`

Add a new header to the system. `hname` is the text name of the header and `ht` is the corresponding compressed header type. If no compressed header type is specified, then `ht` should be -1. The `modinfo` structure defines the dynamic link and load parameters for the local system.

```
int res_query (struct sockt_addr *dest, int ttl, char *query, int size);
```

Send a formulated query that is stored in `query` and is of length `size`. The destination of the query, typically the multicast address 224.100.100.100, is specified in `dest`. The maximum TTL value for the query is specified in `ttl`. The routine will return `E_FOUND` if the query is successful and `E_NOT_FOUND` if it is not. This routine should automatically compress `query` and transmit it.

```
int res_request_get (void *function, int pt);
```

This routine asks the server to send requests, for a specific payload type, `pt`, to a specific `function` for processing. If the value for `pt` is negative, then the request is an ALRP request. Otherwise, it is an ATP request. Multiple functions may be daisy chained. If a function does not process the request, it must call `res_request_return ()` to allow another routine to process the data. This function is used in tandem with active network operating system features. Specific processing for particular functions are described below and in Appendices B and C. Payload types are also described in Appendices B and C.

```
int res_request_return (int pt, char *data, long size);
```

This routine returns a request, described by the payload type, `pt`, the data itself, `data`, and the length of the data, `size`, to the handler. The request may have been or may not have been processed by the calling routine, e.g., a new request. The handler will then send the request to another function. This function is used in tandem with active network operating system features.

Table 10. Resolution Handler Control Options

Option	Description
ALRP_ROUND_TIMEOUT	Time, in seconds, for the scaled delay at the end of each ALRP round.
ALRP_MAX_TTL	System limit on the maximum TTL for each query and announce.
ALRP_SERVER	Boolean value determining if the node should respond to query requests.
ALRP_SERVER_LIST	List of nodes that the server should respond to.
ALRP_PROXY_SERVER	Boolean value determining if the node should become a proxy-server.
ALRP_PROXY_LIST	List of value nodes that the server can proxy for.
ALRP_CACHE_*	Cache control parameters. Individual parameters not yet known.
ALRP_COST_MAX, ALRP_COST_LOW	Maximum and minimum cost ranges for default searches.
ALRP_PAYMENT_TYPE	Currency type that node prefers to use.
ALRP_PAYMENT_TYPE_ACCEPT	Currency type that node will accept.
ALRP_PAYMENT_ACCOUNT	Account information that node will use for library payment.
ALRP_PUBLIC_KEY_*, ALRP_PRIVATE_KEY_*	Various public and private keys for different cryptographic algorithms.
ALRP_PORT_QUERY, ALRP_PORT_ANNOUNCE, ATP_PORT	Default ports for the protocols.
ALRP_ADDR_QUERY, ALRP_ADDR_ANNOUNCE	Default addresses for the ALRP actions.

`int res_ctrl (int option, char *value, int size);`

Set and get system options, such as the default timeouts, restrictions, and so forth. This function is used in tandem with active network operating system features. An incomplete list of possible options is provided in Table 10.

`int res_ping (SOCKTYPE *dest, char *data, int *size);`

Send an ATP Ping request to the destination node `dest` and wait for the return `data`. The length of the data is returned in `size`. The receiver processes the request by placing standard ping information in the return data.

`int res_get (SOCKTYPE *dest, char *name);`

Get active code from a node, specified by `dest`, in the network. The URL of the active code is provided in `name`. The URL path can either be the original specific name, the specific name, or the generic name of the library. Original specific names will include a hostname that is not the name of the destination host. Note that the specific names are separated by the generic name by the leading "active:\\" field. The receiver processes the request by locating and transferring the requested code. If the code is not found, an `E_NOT_FOUND` error is returned.

`int res_put (SOCKTYPE *dest, char *name);`

Deliver active code to a network node, specified by `dest`. The local name of the active code is provided in `name`. The code is assumed to be stored on the local system, which means that the data and headers are indexed to the local name. The receiver will receive the code and store it into its cache. The receiver has the option of discarding the code or aborting the transfer.

`int res_post (SOCKTYPE *dest, char *name);`

Deliver to and execute active code at a network node, specified by `dest`. The local name of the active code is provided in `name`. A Content-Type header must be provided or the transfer is aborted. The receiver will process the headers and obtain header-based dependencies before attempting to execute the code. The code execution handler may also resolve dependencies during execution of the code.

`int res_remove (SOCKTYPE *dest, char *name, void *hdr);`

Remove active code from a network node, specified by `dest`. The name of the active code is provided in `name` and any required headers are sent in `hdr`. This is a restricted action and authentication may be required. If the library is not found, the receiver should return an `E_NOT_FOUND`.

`int res_dict (SOCKTYPE *dest, char *name);`

Obtain the dictionary specified by `name` from the network node specified by `dest`. The requesting node will automatically insert the received dictionary into the system.

```
int res_redirect (SOCKTYPE *src, SOCKTYPE *dest);
```
Tell a remote, `src`, that it should redirect requests to another node identified by `dest`. This node can be any node on the network. This is a privileged action and the calling function must have correct access privileges, otherwise an `E_ACCESS` error will be returned.

```
int res_announce (SOCKTYPE *dest, char *data, long size);
```
Used by library services to announce the availability of a library, new or old. The destination, `dest`, can be a host or a multicast group. The announcement data is in `data` and its length is in `size`.

```
int res_add_code (struct active_rec *);
```
This function adds the code identified by `active_rec` into the local database. Duplicate code is automatically eliminated by the local system. A detailed discussion on the reception algorithm is Appendix B.3.3.

```
int res_del_code (struct active_rec *);
```
This function deletes the code identified by `active_rec` from the local database. Code in use may not be deleted and will result in an `E_RESOURCE_IN_USE` error.

```
struct active_rec *res_get_code (char *name);
```
This function will attempt to find and return the `active_rec` structure for the code identified by `name`. `name` is the original specific name, the specific name, or the generic name. If the code is not found, `NULL` is returned. If multiple matches are in the database, which may be the case in a name generic search, multiple `res_get_next ()` calls may be used.

```
struct active_rec *res_get_next (void);
```
This routine is called, after `res_get_code ()`, to return the next matching item of code. `NULL` indicates that no more matching code could be found.

The current definition of `struct active_rec` is shown in Figure 22. The structure has pointers to the code and headers, the original source node where the code was obtained, and copies of some headers. The headers for generic name, specific name, and original specific name are used in searches. The headers for content-type and platform are used in code execution. Other headers can be found in the header file.

```
struct active_rec {
      char *name_generic;            // The generic name of the code
      char *name_specific;           // The specific name of the code
      char name_specific_original;  // The specific name of the code
      char content_type;             // File type
      char platform;                 // Platforms the code runs on
      char *fn_code;                 // Filename of code
      char *fn_hdr;                  // Filename of headers
      long index;                    // Local index number
      struct sockaddr_in srcaddr;   // host we got this from
}
```

Figure 22. Definition of active record structure.

Appendix B. Active Transport Protocol

The active transport protocol (ATP) is a simple transaction protocol used to transport active code and support query requests. It is expected that multiple definitions of ATP will be made to support various protocols. These definitions fall into two basic classes, unreliable and reliable transports. The header and base action commands are the same, regardless of the ATP variant. ATP is primarily used as a vehicle to support the active library resolution service and to investigate the functionality that is required of an active code transport service. This appendix presents an overview of the protocol transactions, the unreliable variant, the reliable variant, the ABNF rules, the protocol headers, and translation servers.

B.1 Overview

ATP is a protocol that is based on simple transactions. There are two versions of the protocol specified in this document, the first is an unreliable version that operates over UDP and the second is a reliable version that operates over TCP. The model for the TCP version is HTTP [11]. This section will begin by specifying the language that is used and the basic transactions that ATP can perform. The experimental version of ATP is defined to use UDP and TCP ports 1972.

B.1.1 Requirements

The key words "MUST," "MUST NOT," "REQUIRED," "SHALL," "SHALL NOT," "SHOULD," "SHOULD NOT," "RECOMMENDED," "MAY," and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [132].

B.1.2 Protocol Transactions

As presented in Table 11, there are a number of transactions supported by ATP. These are discussed below. Headers that are referenced in the discussion are presented in Section B.4. Allowed response codes are described in the discussion of each transaction command; however, operational information is discussed in Section B.1.8.

B.1.3 Acknowledgment/Negative Acknowledgment

The acknowledgment is used solely to let the source know that the transmission has been received. This is, in network parlance, a positive acknowledgment. The negative acknowledgment forces the source to retransmit packets starting from the included sequence number. A negative acknowledgment, however, acknowledges receipt of packets prior to the specified sequence number. The acknowledgment actions are only used if an unreliable protocol is used.

Table 11. ATP Protocol Transactions

Payload Type	*Meaning*	*Description*
0	Ack/Nack	The Ack action provides acknowledgment information for transmitted data. The payload flags distinguish an acknowledgment from a negative acknowledgment (Nack). Applicable only for unreliable variants.
1	Ping	The Ping action is used to determine the remote implementation and the level of support present on the remote node.
2	Get	The Get command is used to retrieve one specified active code.
3	Put	The Put payload type is used to transfer one specific active program or active library. Repeated Put actions must be used to transport multiple modules of code. Note that a Payload Flag must be set for Put to execute the transmitted active code.
4	Remove	Remote clients and servers may wish to delete active code from nodes on the network. This can be accomplished be using the Remove action and transferring the identifying headers.
5	Response	The Response action returns status information about the request and any required data.
6	Status	The Status command forces a status response from the remote node for an outstanding request. Applicable only for unreliable variants.
7	Abort	The Abort payload type requests that the remote node abort an outstanding request. If the response to the supposedly aborted request is received after an abort is issued, the client must ignore the response. Applicable only for unreliable variants.
8	Redirect	The Redirect action informs a client that it should access a particular node to perform the request.
9	Quit	The Quit command terminates a session.
10	Dictionary	The Dictionary action is used to transfer compression dictionaries.
11-199	Reserved	These set of actions are reserved for future use.
200-255	User Defined	These set of actions may be defined in a specific application. Ping requests are used to determine the type of server at the remote end.

B.1.4 Ping

The Ping command asks the remote host to respond with the Active-Version, Vendor-Name, Revision, and Accept headers. These headers MUST be transferred and MUST NOT be encrypted. The remote host MAY respond with other vendor supplied headers. Vendor supplied headers MAY be encrypted. Ping has no associated data transmitted with it. Both the request and response MUST be no larger than one ATP packet in the unreliable version. The remote host responds with a 202 Response command followed with the data.

B.1.5 Get

If a particular library is known to exist on a remote host, the Get command can be used to retrieve the library. The Get command MUST specify either a local specific name, the original specific name, or the generic name. Note that the difference between a specific name and a generic name is that the specific name always starts with a leading “active://” in the URL. The remote host must send a Response with an indication of success or failure before the headers are transferred. The success response code is 203 and possible error codes are 103 and 105. Successful responses are followed by data. Receiver and sender algorithms are described later.

B.1.6 Put/Post

The Put command is used to place, but not execute, active code on a remote host. The library is sent prefaced with a set of MIME headers. The minimum required headers MUST include Active-Version, Content-Type, and Name-Specific headers. The Post command is used to place and execute code. Note that active libraries should never be executed as they are not independent executable programs. The remote host must send a Response with an indication of whether or not the local host should proceed. A 201 response indicates that the local host should proceed and a 106 code indicates the detection of a duplicate library. The remote host may allow the local host to proceed and later end the transmission with a Response code indicating that the library was not accepted. Receiver and sender algorithms will be described later.

B.1.7 Remove

To delete active code on a remote host, the local host should use the Remove command. Note that the local host must have the appropriate permissions to remove the active code. Implementations must decide if the host may remove active code that is currently running. Incorrect permissions result in a 105 error being returned to the local host. Otherwise, a 200 response code is returned. User information, which is used to determine access privileges, is transferred in the headers.

Table 12. Response Status Codes

Status Code	*Meaning*
100	Error: Request line format incorrect or bad packet.
101	Error: Header format incorrect.
102	Error: Unsupported option.
103	Error: Requested item not found.
104	Error: System error.
105	Error: Permission denied.
106	Error: Duplicate library.
110	Error: Unknown command referenced.
199	Error: Version mismatch.
200	Status: Command completed okay.
201	Status: Remote host ready for data.
202	Status: Ping information follows.
203	Status: Requested code attached.
210	Status: Command being processed.
300	Warning: Command may fail.
301	Warning: Duplicate library.
399	Warning: Version mismatch.
400	Redirect: Redirect command to specified host.

B.1.8 Response

The Response command is used to acknowledge a completed action or indicate a failure. Receipt of a Response command automatically acknowledges all outstanding unacknowledged transmissions. The Response payload type is a generic command as it responds to ping, get, put, remove, status, and abort commands. The codes that are used are shown in Table 12.

Response codes 100 and 102 are returned if there was a specific problem found in the local host's command. Response code 101 is used to indicate that a bad protocol header was received. Response code 104 is used to indicate that some system error occurred. Multiple responses may be sent. Other response codes are described with their transaction methods.

B.1.9 Status

The Status command is used to force the remote host to provide status information on a pending request. This is only used in the unreliable variant and used to check an outstanding request. If the command was lost or never received by the remote host, it will respond with a 110 error. Otherwise, if the command was completed, a 200 response code will be returned. If the command is still being processed, a 210 response code will be returned.

B.1.10 Abort

The Abort command is used to abort a pending request. This is only used in the unreliable variant and is used to check an outstanding request. The response codes are the same as the Status command, with the exception that response code 105 indicates that the local host may not abort the request.

B.1.11 Redirect

The Redirect command instructs a host to use another server for the specific command that has been issued. Redirects only apply to one specific command and are equivalent to a response code of 400.

B.1.12 Dictionary

The Dictionary command is used to obtain ALRP compression dictionaries. The semantics are otherwise the same as the Get command. ALRP dictionaries are discussed in Section C.3.3.

B.1.13 User-Defined

To invoke user-defined actions, a ping request should be made first and the Vendor-Name checked. Semantics of user-defined commands are, naturally, defined by the users.

B.1.14 Compatibility

Backwards compatibility can be ensured by examining the protocol version information. Forward compatibility is not guaranteed. To help ensure some level of interoperability, all implementations MUST ignore options that they do not understand. Further, all implementations MUST inspect the version number and either abort the request immediately, with a 199 response code, or avoid the use of new extensions. If new extensions must be used, a 399 response code should be sent.

B.2 ATP Over UDP

This section discusses the features and high-level design of ATP operation over UDP. This specification is by no means complete. It is provided so that active code transport investigators can see what concepts may need to be embedded into an active code transport protocol. The section describes the packet format, the receiver and sender state transition diagrams, and then some transaction processing comments.

B.2.1 ATP Packet Format

This section describes the architecture of an ATP variant that uses an unreliable protocol as the delivery mechanism. To achieve reliability, mechanisms similar to those used in the Transmission Control Protocol (TCP) [133] are used. TCP uses a modified selective repeat ARQ scheme while ATP uses a go-back-*n* ARQ scheme. ATP uses a sliding window based error and flow control scheme similar to TCP. To support hardware implementations, a straightforward ARQ retransmission strategy can be used by limiting the window size to one packet. With some additional flags, source addresses, destination addresses, and a hop-count, ATP can become a reliable internet protocol.

As shown in Figure 23, the ATP packet header is of fixed size but the meaning of certain fields change. The ATP packet header is designed so that fields are on octet boundaries, when possible, and the header is on a 32-bit boundary. The fixed size and boundary features are used to simplify hardware processing of packets.

ATP requests and responses may span over an unlimited number of packets if operating over an unreliable protocol. Responses do not have to be immediately sent and multiple requests may be outstanding for any given source-destination pair. Thus, some network-unique identifier must be present. Each request and response can be uniquely identified by a three-tuple, which is the source address, the destination address, and a source identification field (SourceID) found within the ATP packet. The source identification field always refers to the request source, which would be the remote client's SourceID in the case of a server response. This means that the SourceID returned in Ack/Nack, Response, and Redirect payload types are always the SourceID provided by the remote client. Otherwise the three-tuple may not always uniquely identify a request.

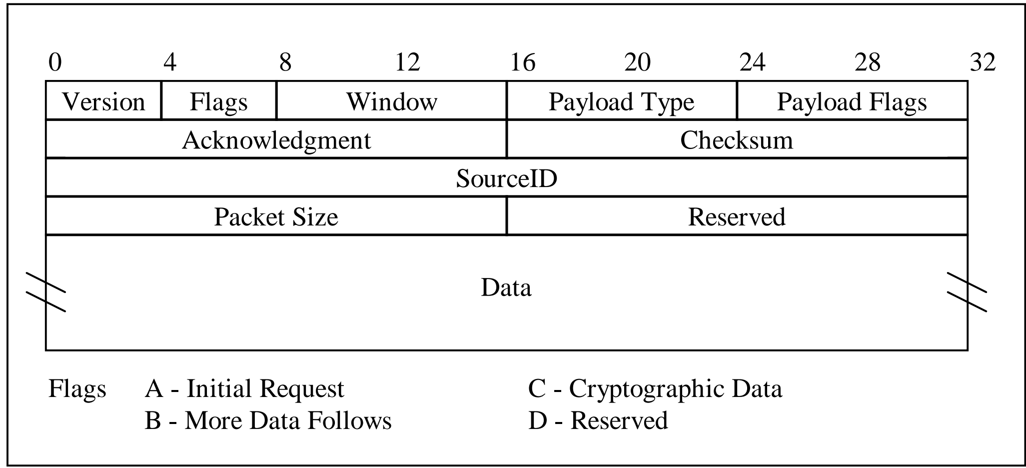

Figure 23. ATP packet format.

An explanation of each field in the ATP header follows. The Version field, which is a 4-bit integer, identifies the version number of this protocol. The current version is version '1', which is represented as the version number minus unity, or zero in binary. The Payload Type is an 8-bit integer and is present in bits 16 through 24 when Flag A, or header bit 4, is set. The payload types are presented in Table 11. The Payload Flags field is an 8-bit flag field, present when Flag A is set, that may be associated with each different Payload Type.

The Payload Flags and Payload Type fields become the Sequence field when Flag A is not set. As stated previously, an ATP packet MUST contain only one request. However, the data for this request can be sent across multiple packets. As discussed earlier, the 32-bit integer SourceID, along with the source and destination addresses, uniquely identifies each request and response pairing. However, some other mechanism must be used to ensure proper ordering of the transmitted data. This order and error control mechanism is the 16-bit Sequence number, which has no direct relationship to the current numbering of bytes being sent. The first packet, where the Sequence field is the Payload Type and Payload Flags field, is automatically assigned sequence number zero. If there is more data, the sequence number is incremented by unity and Flag B is set. Sequence is modulo 65,535. Flag B is set on all packets where there is a following packet of the same SourceID; it is not set on the last data packet. The design of the sequence number has certain security vulnerabilities that can be remedied by the use of cryptographic algorithms.

Window and Acknowledgment are used for flow control and error control, respectively. Window allows a maximum, or window size, of 255 packets to be sent, which is more than sufficient to allow for expiration of duplicate packets in the network. The window size is typically limited by the server, but the client can also limit the window size to a range from one to 255 packets. The 16-bit Checksum is a checksum of both the header and data in the packet. The Packet Size is size, in octets, of the data only. The Reserved field is used for future expansion and for internet-specific variants.

Flag C is used to denote that the payload is either encrypted or has an authentication tag attached to it. Whatever cryptographic action is used, it must apply to the entire data set. Authentication and integrity checks can also include partial header information from the first packet. For the receiver to process the payload, the first 16 bits of the payload on the first packet indicates the cryptographic algorithm. Depending on the algorithm, this may be followed by other required information. The tentative state transition diagram for the protocol operation is included in Section B.2.2.

Note that the SourceID may be internally used by the ATP interface primitives discussed in Appendix A.3. A SourceID of zero may never be transmitted as it is defined to mean that a program has no source. A SourceID should not be reused, even if the same request is repeated.

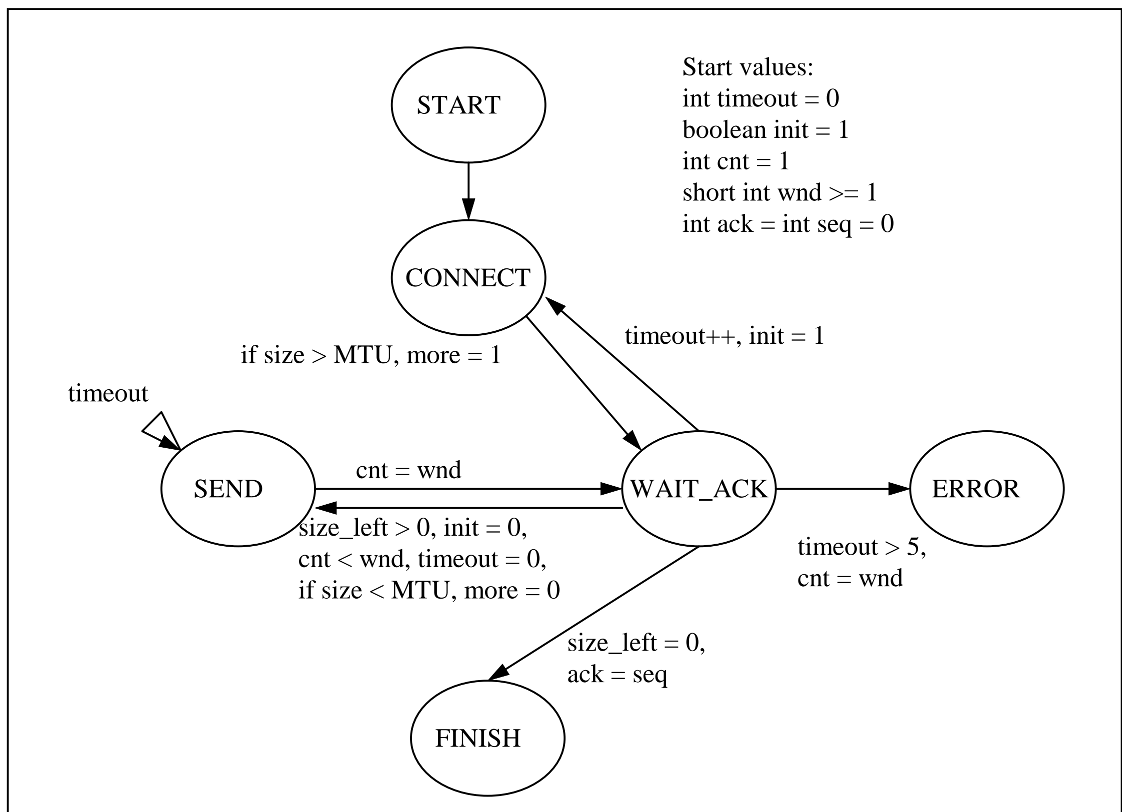

Figure 24. Sender state transition diagram.

B.2.2 State Transition Diagrams

Figure 24 and Figure 25 describe the algorithm of the transport protocol. Note that these state transition diagrams may have clarity and minor design problems. They are included to allow the reader to obtain an understanding of the complexity of the unreliable variant of ATP. These state diagrams are only used if an unreliable transport is used as the underlying protocol. Otherwise, ATP is a simple one transaction protocol that operates on top of a reliable transport protocol.

Figure 24 shows the client, or sender, state diagram. In the CONNECT state, the sender transmits the initial ATP packet that sets up the reliable session. The WAIT_ACK state is the state that waits for acknowledgments to be received. The SEND state transmits packets to the destination. The number of packets that may be sent before returning to the WAIT_ACK state is defined by the window size. When all packets and acknowledgments have been received, the sender enters the FINISH state, which means the connection is terminated. Note that the sender may then enter the receiver state of Figure 25 to receive responses from the server.

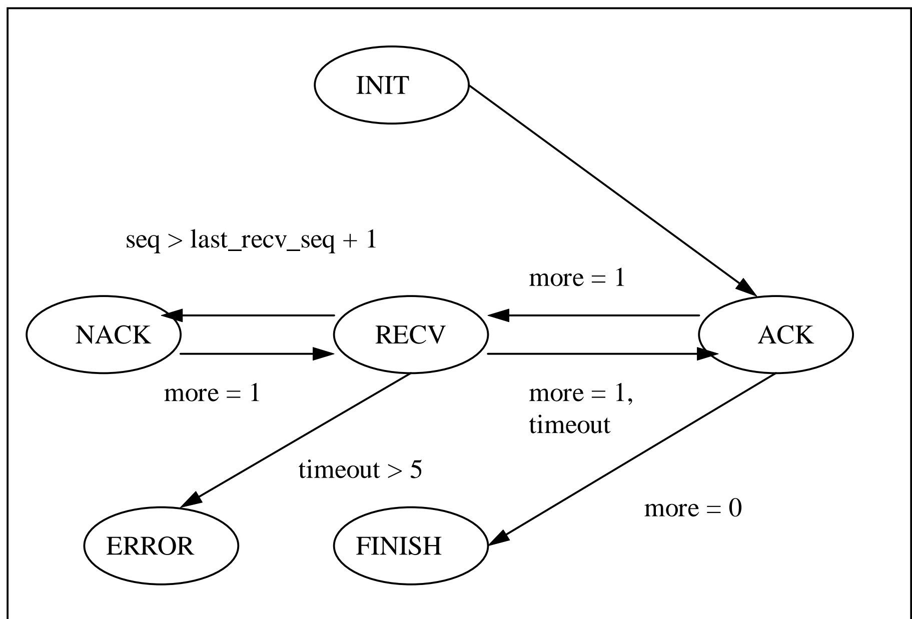

Figure 25. Receiver state transition diagram.

The server, or receiver, state diagram is shown in Figure 25. As can be seen from a comparison of the sender and receiver state diagrams, the complexity needed to achieve reliable transport is biased towards the client. This allows a hardware implementation to simplify the states of Figure 25 and result's in a lower memory requirement. The receiver starts in the INIT state, which is entered when an ATP connection packet is received. This packet is acknowledged in the ACK state and then the receiver immediately moves to the RECV state to wait for more packets. The NACK state is entered only if a packet was discarded in route or not received. The receiver enters the FINISH state only when an ATP packet is received without the more data flag set. The ERROR state is entered only when the connection has not been terminated and data has not been received for a long period. The receiver may enter the sender state of Figure 24 if a response needs to be sent.

B.2.3 Transaction Processing

For the Put and Post commands, the two most significant Payload Flag bits have special significance. The most significant bit is used to denote that the code should be executed upon receipt. The next most significant bit is used to indicate that a response is not required. The

unreliable ATP variant does not require a response to be sent for any command. For the Response command, the 32-bit SourceID is the SourceID of the original request and is used by the receiver to differentiate between Response packets. In the Response packet, the initial Payload Flag field is an 8-bit integer that defines a status code.

B.3 ATP Over TCP

The operation of ATP over TCP, or the ATP reliable variant, is similar to HTTP. ATP should be a low overhead and simple transaction protocol. A significant amount of design and implementation discussion may be found in RFC 2068 [11]. This section discusses the basic transmitter and receiver algorithms. The transmitter is considered to be the client and the receiver is considered to be the server. From a practical standpoint, both nodes are actually servers but for the specific transaction, one node is a client and the other node is a server.

B.3.1 Message Format

The ABNF rules, which are described in Section B.4.1, for the ATP message format is given below.

```
ATP-Request          = Action-Type [Name] "ATP/1"
  Action-Type        = "Ping" | "Get" | "Put" | "Remove" | "Response" |
                       "Redirect" | "Dict" | Extension-Token
  Name               = Active-URL | Name-String
```

The transmitter may use all actions except for Redirect. The receiver may only use the Response and Redirect actions. Specific algorithms are described in the next two sections.

B.3.2 Transmitter Algorithms

The transmitter sends a request based on the message format discussed in Section B.3.1. The general algorithm used by the transmitting process is given below.

1. Send the request.
2. Wait for response message indicating:
- Data will be sent (Get, Ping, and Dict requests). Go to step 3.
- The remote is ready for data to be sent (Put and Post requests). Go to step 4.
- The remote has completed the action or the action failed (Remove request). Go to step 5.
3. Wait for headers.
- Receive, and decompress if possible, headers until a null header is received. Process transmission related headers such as Content-Length. Content-Type, Content-Transfer-Encoding, etc.
- If there is no Name-Specific-Original header, translate the Name-Specific header into the original specific name.
- Receive data and verify Content-Length field, if supplied.

- Optionally wait for a successful transfer response from remote. Go to step 6.
4. Send headers, compressed if necessary. Provide appropriate transmission related headers when possible.
- Send data.
- Optionally, send data transmitted Response. Go to step 6.
5. Wait for response from remote.
6. If no more requests, quit.
7. Check for a ready response message from the remote.
8. Process next command and return to step 1.

The process is for the transmitter to perform one of three actions. The first action is to send the request, wait for the headers and the data, and either quit or process the next action. The second action is to send the request, headers, and data, then quit or process the next action. The third action is to send the request, verify that request was completed, and then quit or process the next action. A number of possible response verification steps may be performed or ignored. If more than one request is being sent in a session, the optional steps MUST be performed.

B.3.3 Receiver Algorithms

The receiver algorithm is the reverse of the transmitter algorithm. It is given below.

1. Wait for a request and spawn a new process or thread.
2. If the request is for a file:
- Verify that the file is present on the local system, otherwise send the remote an error.
- Inform the remote that data will be sent via a Response command.
- Send the headers for the file. Provide Content-Length and Content-Type information when possible.
- Send the data for the file. Go to step 5.
3. If the request is to receive a file:
- Verify that the remote can send the file.
- If not, send an access denied Response command and go to step 5.
- Send a Response command indicating that the local node is ready to accept the file.
- Wait for the headers and then the data.
- If there is no Name-Specific-Original header, translate the Name-Specific header into the original specific name.
- If the code is a duplicate copy, discard it and send a duplicate code Response command and go to step 5.
- Indicate successful reception. Go to step 5.
4. If the request is a generic action:
- Verify that the remote has the authorization to perform the action.
- If not, send an access denied Response command and go to step 5.
- Perform the action.
- Send an action completed Response command.

5. Either close the connection or wait for another request.

The receiver simply processes the request by either 1) sending the requested file with headers, 2) receiving the headers and the file, or 3) processing the command. When receiving the file, the receiver must test to see if duplicate code has been received. This can be performed by verifying that data and headers exactly match, with the exception of the Name-Specific header, another active program stored on the local node. Exactly matching headers and data means that there was strict duplication. Loose duplication may or may not be permitted by the local node. Loose duplication means that some part of the headers match but others do not — the active code itself exactly matches. If the only header that varies is the Name-Generic, Name-Specific, or Name-Specific-Original headers, then the code should not be considered to be duplicate code. All the different headers should be stored, however. If the active code is different but the headers are the same, the code is not duplicate. How the local node handles duplication is defined by the implementation.

B.4 Header Rules

This section discusses the official ATP headers. Some headers are borrowed from HTTP and specific technical detail may be found in RFC 2068 [11]. The operation of search targets and constraints are discussed in Appendix C.

B.4.1 Augmented BNF Rule Set

The content of this section is verbatim from RFC 2068 [11] with some textual reformatting. All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (ABNF) [102]. Implementers need to be familiar with the notation in order to understand this specification. The ABNF includes the following constructs. Implicit whitespace is allowed.

name = definition
The name of a rule is simply the name itself (without any enclosing “<” and “>”) and is separated from its definition by the equal “=” character. White space is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as SP, HT, CRLF, DIGIT, ALPHA, etc.

“literal”
Quotation marks surround literal text. Unless stated otherwise, the text is case-insensitive.

rule1 | rule2
Elements separated by a bar (“|”) are alternatives, e.g., “yes | no” will accept yes or no.

(rule1 rule2)
Elements enclosed in parentheses are treated as a single element. Thus, “(elem (foo | bar) elem)” allows the token sequences “elem foo elem” and “elem bar elem”.

*rule
The character “*” preceding an element indicates repetition. The full form is “<n>*<m>element” indicating at least <n> and at most <m> occurrences of element. Default values are 0 and infinity so that “*(element)” allows any number, including zero; “1*element” requires at least one; and “1*2element” allows one or two.

[rule]
Square brackets enclose optional elements; “[foo bar]” is equivalent to “*1(foo bar)”.

N rule
Specific repetition: “<n>(element)” is equivalent to “<n>*<n>(element)”; that is, exactly <n> occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3ALPHA is a string of three alphabetic characters.

B.4.2 Basic Rules

Material in this section is from RFC 2068 [11] with minor modifications. These rules describe basic parsing constructs. The US-ASCII coded character set is defined by ANSI X3.4-1986 [105].

OCTET = <any 8-bit sequence of data>
CHAR = <any US-ASCII character (octets 0 - 127)>
UPALPHA = <any US-ASCII uppercase letter “A”..“Z”>
LOALPHA = <any US-ASCII lowercase letter “a”..“z”>
ALPHA = UPALPHA | LOALPHA
DIGIT = <any US-ASCII digit “0”..“9”>
CTL = <any US-ASCII control character (octets 0 - 31) and DEL (127)>
CR = <US-ASCII CR, carriage return (13)>
LF = <US-ASCII LF, linefeed (10)>
SP = <US-ASCII SP, space (32)>
HT = <US-ASCII HT, horizontal-tab (9)>
<"> = <US-ASCII double-quote mark (34)>

The CRLF sequence is the end-of-line market for all protocol elements except the entity-body. The end-of-line marker within an entity-body is defined by its associated media type.

CRLF = CR LF

B.4.3 Extension Rules

An Extension-Token is a placeholder token that can be used to extend the rule under discussion. An Extension-Set is a set of extension tokens that can be defined to extend the rule under discussion. Numeric comparisons are allowed only if the rule meets the constraints of Section C.3.2; otherwise, regular expression comparisons are used.

B.4.4 Basic Date and Time Rules

The RFC1123-Date is a timestamp that is defined in RFC 1123 [101]. It MUST be used since no other data formats are supported.

RFC1123-Date = WkDay "," SP Date1 SP Time SP "GMT"

The compressed format of the RFC1123-Date, which is dictionary code number 130, is a set of octets as below, where DAY, MONTH, HOURS, MINUTES, and SECONDS, are unsigned 8-bit integers and YEAR is an unsigned 16-bit integer. The conversion to GMT is assumed.

<130> DAY MONTH YEAR HOURS MINUTES SECONDS

The RFC1123-Date is not usable as a search target or constraint by itself.

B.4.5 Basic Domain-Set Rule

The Domain-Set rule defines one or more Internet domains.

Domain-Set = Domain-Name *("," Domain-Name)

There is no compression rule for Domain-Set and usage as a search target or a constraint is not applicable.

B.4.6 Active-Version Rule

The Active-Version rule defines the version of these headers. It is MUST be present in all implementations that conform to these specifications. The current version is "1.0".

Active-Version = "Active-Version" ":" 1*DIGIT "." 1*DIGIT

The compressed header is of the following format, where MAJOR and MINOR are 8-bit integers. The Active-Version header may not be used as a search target or constraint.

<129> MAJOR MINOR

B.4.7 Internet Media Types

The new Internet media type of `active` is used to denote all active code. With the exception of some tokens used for transfer encodings, the subtypes are the various active programming languages. Note that if the `octet-stream` subtype is used, a platform header MUST be provided.

Active-Type = “active”
Active-Subtype = “octet-stream” | “text” | “unicode” | “C” | “java” | “C++” | “netscript” | “perl” | “pascal” | “TCL” | Extension-Token
Type = Active-Type | Extension-Set
Subtype = Active-Subtype | Extension-Set

The values for the Extension-Set are for the other MIME content types, as defined by IANA. Note that ATP does not currently support multipart media types. The media types may not be directly used as a search target or constraint.

B.4.8 Accept Rule

Accept operates in a similar manner to the HTTP definition. Note that if the wildcard, or ‘*’, media type is used, then the receiver is specifying that it will attempt to find a interpreter or compiler that supports the transported type. Compression rules are defined after the standard rule. Accept headers may not be used as a search target or a constraint.

Accept = “Accept” “:” “*/*” | (Type “/” “*”) | (Type “/” Subtype) [“,” (Type “/” Subtype)]
= <139> (<136> | Type) “/” Subtype [“,” Type “/” Subtype)]

B.4.9 Content Rules

There are a number of rules that describe the basic content of the data being transferred. It is recommended that all of these rules be used in ATP.

Content-Transfer-Encoding
= “Content-Transfer-Encoding” “:” Type “/” Subtype
= <140> <136> “/” Subtype

Content-Transfer-Encoding is used to determine how the data is being sent. If it is not present, the assumption is US-ASCII. The only allowed transfer encodings are `active/octet-stream`, `active/text`, `active/unicode`, and `active/base-64`. Specific details of each transfer encoding can be found in [11]. Transfer encodings may not be used as a search target or a constraint.

```
Content-Length              = “Content-Length” “:” 1*DIGIT
                            = <141> LENGTH
```

The Content-Length defines the number of bytes of data. If this is not provided, then the message length is taken from the amount of data that is sent by ATP. Applications SHOULD compare the Content-Length, if provided, against the received size. The compressed form is straightforward and included above. The LENGTH field is a 32-bit unsigned integer value. Content lengths may be used as a search target or a constraint. Additionally, any valid numeric comparison may be performed on the LENGTH field.

```
Content-Type                = “Content-Type” “:” Type “/” Subtype
                              [“;” Program-Type]
                            = <142> (<136> | Type) “/” Subtype [“;” Program-Type]
  Program-Type              = “Interpreter” | “Compiler” | “Library” | “Handler” |
                              “Program” | “Device” | Extension-Token
```

Content-Type is used to denote the type of data being transferred. Under ATP, the only data transferred should be some form of an `active` media type. The Program-Type rule helps to classify the active code. This is especially important when an operating system is seeking a specific software type, such as a compiler for active code. It is recommended that all active code provide this header. If it is not present, then the assumption is that the Program-Type is `Program`. Content types may be used as a search target and constraint using regular expressions.

B.4.10 Naming Rules

There are three library names that can be used, the Name-Specific, Name-Specific-Original, and the Name-Generic identifiers. The function and purpose of these headers are described in Section 4.3.3.

```
Name-Specific               = “Name-Specific” “:” Active-URL
                            = <133> Active-URL
Name-Specific-Original      = “Name-Specific-Original” “:” Active-URL
                            = <134> Active-URL
  Active-URL                = (“active:” “//” Host [“:” Port] [Absolute-Path]) |
                              (“none:” User-String)
```

Both the Name-Specific and Name-Specific-Original can be used as search targets and constraints using regular expressions.

Note that, in the Name-Generic rule, the form “@” SourceID, where SourceID is converted into a text representation of the 32-bit integer, is a special context that allows an active program to delete itself or other active programs.

```
Name-Generic           = “Name-Generic” “:” Name-String
                       = <132> Name-String
Name-String            = (“MediaType@” SubType) | ((Token | Quoted-String)
                         [“@” Context-String [“@” Scope]])
  Context-String       = Token | Quoted-string
  Scope                = “IANA” | “ISO” | “IEEE” | “ANSI” | “Global” | “Site” |
                         “Subnet” | “IP” “=” IP-Hop-Limit | Domain-Set |
                         Extension-Token
```

The Name-Generic rule can be used as a search target and a constraint using regular expressions. For most applications, it is the usual search target.

B.4.11 Platform and Revision Rules

Two headers that should be provided for any active code are the Platform and Revision headers.

```
Platform               = “Platform” “:” Vendor “-” Processor “-”OS “-” Version
                       = <143> Vendor “-” Processor “-”OS “-” Version
  Vendor               = “Sun” | “All” | “Unknown” | Extension-Token
  Processor            = “SPARC” | “x86” | “All” | Extension-Token
  OS                   = “Solaris” | “Linux” | “All” | Extension-Token
  Version              = “All” | Extension-Token
```

For the Vendor, Processor, OS, and Version rules, hyphens MUST NOT be used in the string. The Platform rule may be used as a search target and a constraint using regular expressions. Individual field comparisons are used when the “All” token is present in a field. The “All” token denotes that all vendor types, for example, are valid.

```
Revision               = “Revision” “:” 1*DIGIT “.” 1*DIGIT
                       = <144> MAJOR MINOR
```

The Revision rule may be used as a search target and a constraint. Note that the Revision rule defines a standard version numbering scheme for active networks. This allows numeric comparisons to be made for each field. The two 8-bit integer fields are the MAJOR and MINOR revision levels.

B.4.12 Miscellaneous Information Rules

A set of generic information about the active code can also be embedded. This includes basic information about the author, which may be an entity, and creation dates.

```
Code-Author            = “Author” “:” Name “(” Email “)”
                       = <145> Name “(” Email “)”
  Name                 = Token | Quoted-String
  Email                = Token “@” Host
```

The Code-Author rule can be used as a search target and a constraint using regular expressions. Note that an email address MUST be provided if this field is used. Ideally, this should be a valid email address.

```
Create-Date              = "Create-Date" ":" RFC1123-Date
                         = <146> RFC1123-Date
Revision-Date            = "Revision-Date" ":" RFC1123-Date
                         = <147> RFC1123-Date
```

Both date rules can be used as search targets and constraints using regular expressions as well as numeric tests.

```
Vendor-Name              = "Vendor-Name" ":" Token | Quoted-String
                         = <131> Token | Quoted-String
```

The Vendor-Name rule can be used as a search target and as a constraint using regular expressions.

B.4.13 Signature Rule

The Signature rule provides an integrity and/or authentication check, depending on the cryptographic algorithm used, for the active code. The Signature rule may be used as a search target and a constraint using regular expressions.

```
Signature                = "Signature" ":" Crypto-Method [ ";" "Encoding" "="
                           Transport-Encoding ] ";" "Key" "=" Crypto-Key
                         = <148> Crypto-Method [ ";" "Encoding" "="
                           Transport-Encoding ] ";" "Key" "=" Crypto-Key
Crypto-Method            = "MD5" | Extension-Token
Crypto-Key               = Token | Encoded-Token
```

A related HTTP header is the Content-MD5 header. The Signature header provides a generic integrity and authentication mechanism that is independent of any particular cryptographic algorithm.

B.4.14 Interface Rules

A number of rules exist to support data type definitions and dependency information. Different rules must be defined for different languages. The rules given below specify values for the C language.

```
C-Interface-Parameter    = (["volatile" | "const"] ["auto" | "register" | "extern" |
                           "static"] ["signed" | "unsigned"] "char" ["*"]) | (["volatile"
                           | "const"] ["auto" | "register" | "extern" | "static"]
```

["signed" | "unsigned"] ("short int" | "short") ["*"]) |
(["volatile" | "const"] ["auto" | "register" | "extern" |
"static"] ["signed" | "unsigned"] "int" ["*"]) | (["volatile" |
"const"] ["auto" | "register" | "extern" | "static"] ["signed"
| "unsigned"] ("long" | "long int") ["*"]) | ("void" ["*"]) |
(["volatile" | "const"] ["auto" | "register" | "extern" |
"static"] "float" ["*"]) | (["volatile" | "const"] ["auto" |
"register" | "extern" | "static"] "double" ["*"]) | (["volatile"
| "const"] ["auto" | "register" | "extern" | "static"] "long
double" ["*"]) | ("@" Composite-Name ["*"])

C-Composite-Type = "typedef" | "struct" | "union" | "enum" |
Extension-Token

The Composite-Primitive rule defines a composite data structure. Note that duplicate Composite-Names are not permitted. Applications should automatically generate a composite name as the name itself is not important, it simply is used to provide an appropriate reference for Interface-Primitive and Interface-Dependence headers.

Interface-Parameter = "Internal" | C-Interface-Parameter | "@"
Composite-Name | Extension-Set

The Interface-Parameter rule is used to define primitive types ofo a language as well as system-dependent internal data structures, as represented by the "Internal" keyword. These system-dependent structures include, but certainly are not limited to, file handle structures and other I/O control structures. The search assumption is that the functions operate in the same manner, regardless of the environment, and the only differences are system-dependent structures. Platform matching should appropriately resolve the system dependent structures.

Composite-Primitive = "Composite" ":" Sub-Type ";" Composite-Type ";"
Composite-Name ";" *(Interface-Parameter ",")
= <150> Sub-Type ";" Composite-Type ";"
Composite-Name ";" *(Interface-Parameter ",")
Composite-Type = C-Composite-Type | Extension-Set
Composite-Name = Token | Quoted-String

The Interface-Primitive is used to specify an external interface that is available in the code. The Interface-Primitive and Composite-Type rules may be search targets and constraints through the use of regular expressions. The Interface-Primitive rule is also used to define global constants and other globally accessible variables.

Interface-Primitive = "Interface" ":" Sub-Type ";" Symbol-Name [","
Symbol-Type] ";" [Return-Value *("," Return-Value)] ";"
[Passed-Value *("," Passed-Value)] [";" Call-Type]

```
                        = <149> Sub-Type ";" Symbol-Name ";" [Return-Value
                          *("," Return-Value)] ";" [Passed-Value *(","
                          Passed-Value)] [";" Call-Type]

Symbol-Name             = Token | Quoted-String
Return-Value            = Interface-Parameter | Extension-Set
Passed-Value            = Interface-Parameter | Extension-Set
Call-Type               = "C" | "Pascal" | Extension-Token
```

The Interface-Dependence rule is used to denote the libraries upon which the active code depends. This rule may be a search target and a constraint through the use of regular expressions.

```
Interface-Dependence    = "Dependence" ":" Name-String | Source-URL ";"
                          Sub-Type ";" Symbol-Name ";" [Return-Value *(","
                          Return-Value)] ";" [Passed-Value *("," Passed-Value)]
                          [";" Call-Type]
                        = <150> Name-String | Source-URL ";" Sub-Type ";"
                          Symbol-Name ";" [Return-Value *("," Return-Value)] ";"
                          [Passed-Value *("," Passed-Value)] [";" Call-Type]
```

B.4.15 Encryption Rules

The Encrypted-Header embeds another ATP header within itself. The embedded header is encrypted and must be decoded. The encrypted header typically provides payment information. The Encrypted-Header itself may not be a search target, but the embedded header may be a search target.

```
Encrypted-Header        = "Encrypted" ":" Crypto-Method [";" "Key" "="
                          Crypto-Key ";" Encoding-Type ";" Encoding-Size]
                        = <152> Crypto-Method [";" "Key" "=" Crypto-Key ";"
                          Encoding-Type ";" Encoding-Size]
```

The Encrypted-Data header is used to inform the receiver that the data that was transported is encrypted. Information about the cryptographic algorithm and any public keys are provided by the header. This header may not be a search target.

```
Encrypted-Data          = "Encrypted-Data" ":" Crypto-Method [";" "Key" "="
                          Crypto-Key]
                        = <155> ":" Crypto-Method [";" "Key" "=" Crypto-Key]
```

How keys are exchanged has not been defined.

B.4.16 Payment Rules

The basic supporting definitions for commercial transactions are given below.

```
Currency-Type          = “US-Dollars” | “Pound” | Extension-Token
Currency-Unit          = 1*DIGIT “.” 2*DIGIT [“.” 1*DIGIT]
Method                 = “VISA” | “American Express” | “MasterCard” |
                         “Discover” | “Bank” | Extension-Token
```

The Payment rule, which may not be a search target, is used to indicate how the requester will pay to access the particular active code being requested. It can only be used in an ATP get request. Typically this header is encrypted using the Encrypted-Header rule.

```
Payment                = “Payment” “:” Method “;” Account “;” Currency-Type
                         “;” Currency-Unit
                       = <155> Method “;” Account “;” Currency-Type “;”
                         Currency-Unit
```

The following headers are used by an active library source to denote acceptable forms of payment and the cost to use the library. Payment-Accept may only be used as a search target and a constraint that allows regular expressions whereas Usage-Cost may be compared using numeric tests on the Currency-Unit value.

```
Payment-Accept         = “Payment-Accept” “:” Method [“;” Currency-Type]
                       = <156> Method [“;” Currency-Type]
Usage-Cost             = “Usage-Cost” “:” Currency-Type “;” Currency-Unit
                       = <157> Currency-Type “;” Currency-Unit
```

B.4.17 Authentication Rules

Authentication is performed using the WWW-Authenticate header. Operational characteristics of this header may be found in RFC 2068 [11].

B.4.18 Restriction Rule

A number of distribution restriction rules may also be applied. These rules are used to limit where active code can be sent. These headers may be used as a search constraint through the use of regular expressions.

```
Restriction            = “Restrict” “:” Restriction-Options *(“;”
                         Restriction-Options)
                       = <158> Restriction-Options *(“;” Restriction-Options)
  Restriction-Options  = (“Domain” “=” Domain-Set) | Extension-Option
```

B.4.19 Pragmas

Pragmas are directives to a server and are of the form given below. Pragmas may not be used as a search target or constraint.

```
Pragma                    = “Pragma” “:” Pragma-Rule
                          = <135> Pragma-Rule
```

The following pragma statement instructs a node not to cache the active code being delivered.

```
No-Cache                  = “No-cache”
```

To further support backwards compatibility, the no compression pragma has been defined. This pragma must appear as the first header in the active header set.

```
No-Compression            = “No-compression”
```

The Prediction pragma statement is used to denote access information about the active code that was obtained from the source of the transmission. This information can be used by the recipient in a decision to cache the active code for future requests. Precise semantics were not investigated.

```
Prediction                = (“Predict” “=” 1*DIGIT “.” 4*DIGIT [Active-URL]) |
                            (“Statistic” “=“ 1*DIGIT “/” 1*DIGIT [Active-URL])
```

The Multi-Response pragma statement indicates that a server has multiple matching libraries. The number of libraries, and if the server will directly send the libraries, is indicated by the header.

```
Multi-Response            = (*DIGIT “;” [“Will Send”])
```

B.5 Translation and Proxy-Servers

There are a number of potential intermediate nodes that can process ATP requests. These intermediate nodes take form as translation and proxy-servers. Translation servers take an ATP request from one internet protocol and send it out, without tunneling, over another internet protocol. Data is not modified by a translation server. Proxy-servers process ATP requests on behalf of a host and may modify the data in transit. The basic operation of a translation server is for the server to wait for a complete request to be received and then take the request and retransmit it over another network. Translation servers are used instead of tunneling to reach a node that is only reachable through another protocol. The specifications of translation servers are not defined in this research.

Appendix C. Active Library Resolution Protocol

The basic operation of the active library resolution protocol (ALRP) is discussed in Chapter 4. This appendix provides details on ALRP operation, the packet formats used by ALRP, the operation of a proxy-server, the client and server algorithms, the query language, the regular expression formats, and potential compression techniques.

C.1 ALRP Operation

ALRP is a simple UDP-based protocol that uses expanding-ring multicast. One request, which may be from one to 64 packets in length, will be multicast on the network at a TTL of one. This means that all nodes on the local network and only on the local network will receive the multicast, assuming that they are in the correct multicast group. If no response is received, a higher TTL is used, which includes more adjacent networks and nodes. If a response is still not received, a higher TTL is used on the next attempt. This process is repeated until a response is received or a timeout condition is met. The remainder of this section will discuss the packet types, MTU issues, and proxy-servers.

C.1.1 ALRP Packet Types

There are two ALRP packet types and four functions that can be performed by the two packet types. The first 12 bytes of each packet type are identical, as shown in Figure 26. The version field is an integer value that identifies the current version of ALRP minus one, which is currently defined to be a binary zero. The current ALRP version is ‘1’. The PT field indicates the payload type, which can either be a request, an announce, a not found, or a redirect packet. The four types are discussed later. The flags field has three reserved bits and the S bit, which is used to indicate if the sender is a proxy-server. The S bit, at bit 8, is used to help eliminate cyclic references to proxy-servers in the situation that there is more than one proxy-server attempting to provide proxy services for the same request. Pragma statements may also be used. The current TTL is equivalent to the TTL that is in the IP header. In other words, it is the TTL being used to transmit the packet. This number will increment as the expanding ring multicast increments. This is included in the ALRP header to simply application processing. The checksum field is the IP checksum [125] mechanism applied only to the header. The source address is the IP address of the local node. The source identifier is a unique 32-bit number generated by the local node. This number is modulo 32 and should be unique over a span of a few minutes. The size of any ALRP packet can be easily obtained by the appropriate socket operation.

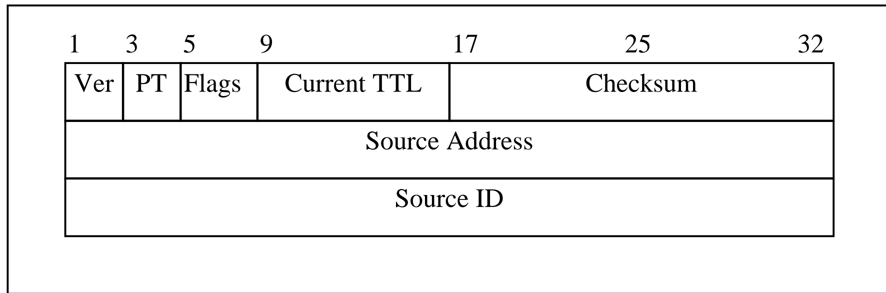

Figure 26. ALRP basic header and announce header format.

The ALRP request, announce, acknowledge, and not found packets use the packet format of Figure 27. The request packet is used to send the query data to all nodes in the multicast group. There can be anywhere from one to 64 packets per request. The announce packet is used to indicate to the source that the library is available at a given node. The announce packet carries data about a specific library and may range from one to 64 packets in length. Thus, the first item that must be in present in the announce packet is the Name-Specific field, followed by any Pragma fields. Pragma options are discussed in Section C.3.1. The acknowledge and not found packets contain no data. The acknowledge packet is used to instruct all servers that the client has found the requested library. The not found packet is used by a proxy-server to indicate that it failed to find the requested library.

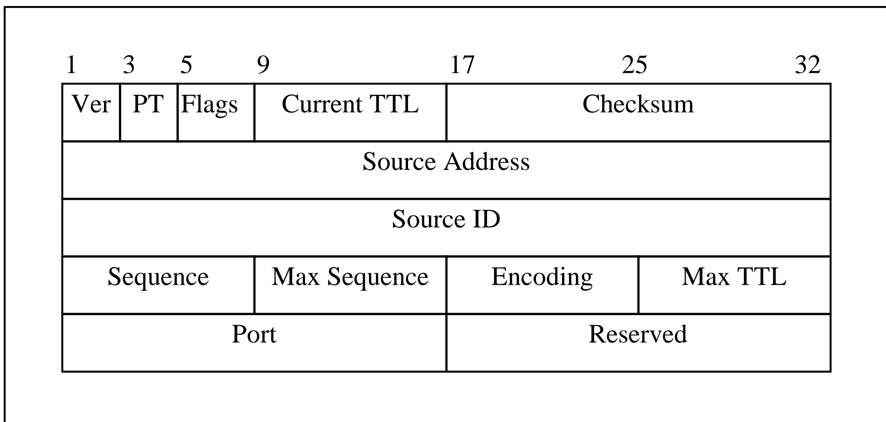

Figure 27. ALRP request header format.

As shown in Figure 27, the only additions to the basic packet format are the inclusion of the current and maximum sequence numbers, the program encoding type, and maximum TTL, and the response port number. The current sequence number, which is unique, identifies the current packet as being the *n*th packet out of a set of packets. The number of packets in the set is identified by the maximum sequence number. The encoding specifies a dictionary encoding scheme that is used on the data. Currently, only two encoding values are defined: binary 0 for no encoding scheme and binary 1 for the C language encoding scheme. The maximum TTL is used to indicate how far away from the source that this request will eventually reach. Thus, if the current TTL of the basic header is equal to the maximum TTL, then the source has reached or is

near to reaching the last round of the expanding-ring multicast. The port number is only used when this packet is a request. The server sends its response to the request directly to the specified port and the source address.

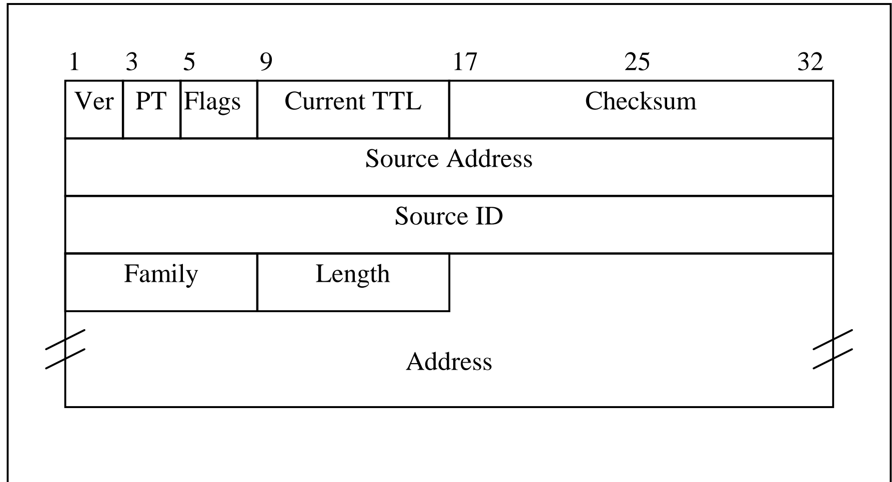

Figure 28. ALRP redirect header format.

The ALRP redirect packet format is given in Figure 28. The redirect command is used to instruct a client to use the specified node as a proxy-server. Proxy-servers are discussed in Section C.1.3. The differences between the redirect packet and the basic packet is that the redirect packet includes the C `sockaddr` structure. This structure contains the family of the address type that is in the packet, the length of the address, and the address itself.

C.1.2 MTU issues

It is possible for the client and server to use the maximum IP packet size of 65,535 to transmit each packet. However, this will result in packet fragmentation. Efficient fragmentation of multicast packets is an issue that is not well understood. Regardless, packet fragmentation results in inefficient network utilization. The typical solution to this problem is to use path maximum transfer unit (MTU) discovery; however, this is not efficiently possible when using multicast. Most implementations will either set the MTU to either be the minimum IP packet size of 576 bytes or equal to the minimum Ethernet packet size of 1,500 bytes minus IP headers. The recommendation, considering that most networks are Ethernet networks, is to use the maximum IP payload size of 1,460 bytes. This is the Ethernet MTU minus 40 bytes of IP and UDP headers.

C.1.3 Proxy-Servers

A proxy-server handles requests on behalf of a client. When a search request is made, the server sends a redirect to the client. The redirect contains the address of the server and informs the client that the server will process the request. The server first checks to see if the library is in its cache. If not, the server will send out a search request via an expanding-ring multicast search. It

may also contact other major servers that it knows about. If the library is found, a success, or announcement, response is then returned to the originating host. If the library is not found, a failure, or not found, response is send to the originating host. Proxy-servers have the option to either direct the client to obtain the library from the proxy-server or the source. This can be done by returning the URL for the library as provided by the original source or as provided by the proxy-server.

C.2 Algorithms

This section describes the control algorithms used by the client and server. The designations client and server are used to indicate which node is transmitting and not their general function. In the normal operation of ALRP, every node is a server that can provide libraries. If a node is requesting a library or is making a new library announcement, it is considered to be a client. Otherwise, it is considered a server.

C.2.1 Client Algorithms

The algorithm for the library announcement is straightforward — the client simply takes the data, divides it into individual packets, and sends each packet until all packets are sent. The client should send the complete announcement at least five and at most ten times to ensure that most, if not all, nodes have received the announcement. The TTL value that the client uses when sending an announcement is implementation or user defined. The client may not exceed five minutes of transmission time.

The following algorithm is used to transmit and process queries. The algorithm simply transmits packets until a response is received or a timeout condition occurs. Proxy-servers require special handling and may also require some user configuration. User configuration is required as some users may not want to use a proxy-server whereas others can only use a proxy-server.

1. Generate a unique sequence number (SourceID) to use for this request.
2. Open and bind a multicast address to a port.
3. Open and bind a port for library announcements.
4. Check to see if the library somehow showed up. If so, exit and report that the library was found.
5. Send the next packet and check for a response on the port bound in step 3.

a) If a library announcement was received, get the library through ATP if it has not already been sent. Verify that the library is the correct library. If it is, multicast an acknowledge packet, exit and report that the library was found.

b) If a proxy redirect was received, wait until a library announcement is returned or a timeout occurs. If the library announcement was received, go to step 5a. If a timeout occurs, go to step 4.

6. If this packet is the last packet of this round:

a) Delay for timeout times the number of packets sent in the round.

b) Increment TTL, resend all packets, starting from the first packet. Go to step 4.

7. If the number of rounds is exceeded, exit and report that the library was not found

Note that clients may not send redirects; however, clients may be configured to ignore redirects.

C.2.2 Server Algorithms

The server algorithm requires that the server record sequence numbers (SourceID) and source addresses for all connections that it receives. A five-minute timer is started at the last received packet for that sequence number and source address. If this timer expires, it is assumed that the client has either found the library, did not find the library, or failed.

1. Wait for a connection request.
2. Spawn a process or thread to service the request.
3. Record the sequence number and source address. The server must ensure that only one process or thread handles packets with the same sequence number and source address.
4. Wait for the next packet.
5. If the packet is an acknowledge packet, exit and ignore the sequence number and source for 300 seconds after the most recently received packet.
6. If the packet number is zero and the receiver does not have a matching library, exit and ignore the sequence number and source for 300 seconds after the most recently received packet.
7. If the packet number is equal to the last packet identifier, see if node has the library in question.

a) If so, send an announcement directly to source.

b) If five minutes have transpired between the first announcement and no message has been received from the source, exit.

c) Wait for an acknowledgment. If no acknowledgments are received within a specified timeout, go to step 7a.

8. If no library was found and the node is a proxy-server, send the source a redirect.
    - Go into source mode and wait until the node obtains a library.
    - If a library was obtained, send it to the source.
    - Otherwise send the source a library not found message.

When the server is checking to see if it has the library, it must use the query and regular expression mechanisms described in Section C.3.2. If a regular expression matches more than one library, the server has a number of actions it can take. The server can send an announcement for one library or for all of the matching libraries. The server should use the multiple match option via the Pragma header. The recommended action is to send an announcement on the most popular library that was matched.

C.3 Query Mechanisms and Language

The query language uses many of the headers presented in Appendix B as search targets or constraints. For certain headers, basic comparisons are defined, including regular expression

matches, greater than, greater than or equal to, less than, and less than or equal to. Each comparison in a query logically ANDed with the preceding query. The basic format of a query is given below. The Header-Rule is any of the searchable headers as discussed in Section C.4.

```
Query-Search            = Header-Rule
```

Server options, constraint categories, and regular expression searches are discussed in this section.

C.3.1 Pragmas

Pragma headers instruct the remote node as to the options that they should use. The following set of headers is defined. The remote node should attempt to satisfy the instructions, but is free to ignore the header.

```
Query-Pragma            = "PRAGMA" SP Parameter
  Return-Type           = "RETURN-TYPE" SP "SEND" | "RETRIEVE"
  Retry                 = "RETRY" SP ("TIMES" SP 1*DIGIT) | ("PERIOD" SP
                          1*DIGIT SP "SECONDS" | "MINUTES")
  Multiple-Match        = "MULTIPLE" SP 1*DIGIT SP "OUT OF" SP 1*DIGIT
  Proxy-server          = "PROXY" SP Hostname
```

The Return-Type pragma instructs the node, during a query, whether or not the client will attempt to retrieve the code or if the server should send the code. The default is for the client to obtain the code. The Retry pragma is an instruction to a proxy-server that overrides the default five-minute expiration between request and announcements. It may also be used to specify the number of attempts. The Multiple-Match pragma instructs the client that there were multiple matching libraries and that the server will be sending information for all matching libraries, individually, to the client. The Proxy-Server pragma is used to record which, if any, proxy-servers processed the request.

C.3.2 Constraint Categories

As discussed earlier, there are a number of comparison operations. These are presented in Table 13. There are seven categories and two are based on regular expression matching. The GNU regular expression definitions [134] are the current specification for ALRP regular expression queries. The first category is a regular expression match and the second category is a regular expression mismatch. The remaining five categories, which are all numerical tests, are discussed below.

Table 13. Query Categories

Categories	*Description*
0	Regular expression match
1	Regular expression not matched
2	Greater than numeric match
3	Greater than or equal to numeric match
4	Equal to numeric match
5	Less than numeric match
6	Less than or equal to numeric match

The mechanism used to perform numerical tests require that the header identifier field be stripped from the header. The header identifier field is everything before and include the colon, ':', in a header. The next field is the field on which the numeric tests are performed on. All other fields are ignored. The value of the field is converted into a floating point number and the query value, found in the request, is compared to the header value found in the local database. For example, given the header `Revision: 1.0`, the header identifier `Revision:` would be removed and the value `1.0` used in numeric comparisons.

Note that numerical date tests vary from the above procedure. The header field and colon are still stripped, but the remaining fields comprise the entire date field and those fields are converted into a single long integer to which the numerical tests are applied. Again, the query date is compared against the date header in the local database.

C.3.3 Dictionary Compression

The following discusses how the interface primitive dictionary compression could be constructed. The dictionary compression mechanism must include support for ATP header compression. First, all possible primitive data types are iterated and assigned a unique index value, from zero to 255. Default variables may or may not be assigned an index. An example of a default variable is the default C definition of "int" being "auto volatile signed int." This comprises the default dictionary. Other compression rules are based on the composite data types. As composite data type names may be reused in interface parameters, compression of these names can become important. Remaining index numbers are assigned to common interface parameters and another dictionary, sent with the data, provide this compression information. Additional information on dictionary encoding techniques may be found in [135].

Appendix D. Experimental and Simulation Data

This appendix contains source data for the figures and tables used throughout the dissertation.

Table 14 presents the source data for the header file analysis of Table 5. The filenames in brackets in Table 14 contain composite data types used by the immediately previous header file. Header files may have more than one set of included composite data types. The counted values for composite data types are approximate; not all possible nested composite data types are included for system data structures. System data structures may be handled using a special escape sequence, as defined in Appendix B.

Tables 15 through 18 provide average values for the three runs for each network case. All tables also show results for the one percent and five percent packet loss cases. Table 15 and Table 16 show the source data comparing the error control design versus the continuous retransmission design of ALRP. Table 15 and Table 16 provide resolution time and packet count data, respectively. Table 17 and Table 18 show the source resolution time and packet counts, respectively. Table 19 and Table 20 provide the source data for the loss analysis, showing resolution and source packet count, respectively.

Table 14. Header File Analysis Source Data

		Function Name Size				Parameters			
Filename	Count	Total	Average	Low	High	Total	Average	Low	High
socket.h	21	150	7.14	4	11	80	3.81	1	6
[in.h]	5	42	8.40	7	11	13	2.60	2	4
resolv.h	14	163	11.64	7	16	55	3.93	0	9
[in.h]	2	18	9.00	7	11	5	2.50	1	4
fcntl.h	4	19	4.75	4	5	9	2.25	2	3
des_crypt.h	2	18	9.00	9	9	9	4.50	4	5
pcap.h	27	339	12.56	9	18	60	2.22	1	5
[pcap.h]	4	44	11.00	8	16	15	3.75	2	7
rpc/auth.h	4	67	16.75	14	23	9	2.25	0	5
[auth.h]	2	17	8.50	8	9	7	3.50	2	5
rpc/clnt.h	13	168	12.92	10	18	44	3.38	1	8
[clnt.h]	5	41	8.20	6	11	18	3.60	1	6
[struct clnt_ops]	6	54	9.00	7	10	16	2.67	0	0
key_prot.h	7	83	11.86	10	15	0	0.00	0	0
pmap_clnt.h	7	81	11.57	8	14	30	4.29	1	10
[pmap_clnt.h]	5	39	7.80	4	11	13	2.60	1	4
pmap_prot.h	2	20	10.00	8	12	4	2.00	2	2
[pmap_prot.h]	3	15	5.00	3	8	11	3.67	1	6
[typedef XDR]	7	75	10.71	8	10	0	0.00	0	0
xdr.h	31	342	11.03	7	18	82	2.65	0	6
[xdr.h]	6	28	4.67	2	11	19	3.17	2	11
[typedef XDR]	7	75	10.71	8	10	0	0.00	0	0
net.h	8	106	13.25	10	17	18	2.25	0	6
[net.h]	5	37	7.40	4	16	41	8.20	2	16
netdb.h	33	365	11.06	6	16	39	1.18	0	4
[netdb.h]	7	45	6.43	3	8	35	5.00	3	8
netdevice.h	39	499	12.79	8	28	47	1.21	0	4
[netdevice.h]	5	44	8.80	5	15	0	0.00	0	0
stdio.h	66	419	6.35	2	12	130	1.97	0	7
bgp_pdu.c	10	151	15.10	10	21	22	2.20	1	5
[bgp.h]	2	20	10.00	10	10	29	14.50	3	26
bgp_util.c	10	180	18.00	7	30	11	1.10	1	3
[bgp.h]	5	44	8.80	7	10	45	9.00	3	26
bgp_sm.c	9	181	20.11	16	24	20	2.22	2	3
[bgp.h]	2	26	13.00	10	16	31	15.50	5	26
bgp_timer.c	2	37	18.50	18	19	2	1.00	0	2
[bgp.h]	2	18	9.00	8	10	35	17.50	9	26
bgp_thead.c	1	21	21.00	21	21	4	4.00	4	4
[bgp.h]	2	18	9.00	8	10	35	17.50	9	26
Function Count	310	3409	11.00	2	30	675	2.18	0	9
Composite Types	115	1065	9.26	2	16	407	3.54	1	26
Totals	425	4474	10.53	2	30	1082	2.55	0	26

Table 15. Error Control Versus Continuous Analysis Source Resolution Time Data

Packet Count	EC		C	
	0.01	0.05	0.01	0.05
1	0.0161	0.0161	0.0208	0.0208
2	0.0235	0.0235	0.0291	0.0291
4	0.0385	0.0385	0.0474	0.0474
8	0.0684	1.9358	0.0862	0.0975
16	0.1281	2.1068	0.1639	0.1864
32	1.1294	3.7289	0.4049	0.4406
64	2.3194	4.1467	0.6989	0.8862

Table 16. Error Control Versus Continuous Analysis Source Packet Count Data

Packet Count	EC		C	
	0.01	0.05	0.01	0.05
1	3	3	8	7.3
2	5	5	9	9
4	9	9	13	13
8	17	415.3	25	27.7
16	33	591.7	49	54.7
32	190.7	2340.3	119.7	126.3
64	431.7	2923.3	200.3	254

Table 17. Source Resolution Time Data

Packet Count	1		2		3		4		5	
	0.01	0.05	0.01	0.05	0.01	0.05	0.01	0.05	0.01	0.05
1	0.0116	0.0116	0.0116	0.0116	0.0208	0.0208	0.0321	0.0321	0.0584	0.0585
2	0.0171	0.0171	0.0171	0.0199	0.0291	0.0291	0.0499	0.0499	0.0936	0.0936
4	0.0280	0.0324	0.0280	0.0280	0.0474	0.0474	0.0871	0.0889	0.1638	0.1638
8	0.0499	0.0499	0.0595	0.0680	0.0862	0.0975	0.1589	0.1589	0.3042	0.3134
16	0.0937	0.1569	0.1154	0.2172	0.1639	0.1864	0.3042	0.3933	0.5851	0.6462
32	0.2625	0.3602	0.2152	0.3026	0.4049	0.4406	0.5950	0.7227	1.2219	1.2289
64	0.4994	0.5975	0.4243	0.6185	0.6989	0.8862	1.2556	1.6827	2.4291	2.5248

Table 18. Source Packet Count Data

Packet Count	1		2		3		4		5	
	0.01	0.05	0.01	0.05	0.01	0.05	0.01	0.05	0.01	0.05
1	4	4	4	4	8	7.3	11	11	20.7	20.7
2	5	5	5	5.7	9	9	15	15	27	27
4	9	10.7	9	9	13	13	24	25.3	43.7	43.7
8	17	17	20.3	22.3	25	27.7	42	42	75.3	80.7
16	33	48.7	38.7	62.3	49	54.7	82	105.7	147.3	167.7
32	82.7	109.3	69	91.3	119.7	126.3	162	194	315	315.3
64	146	176	133.3	191.3	200.3	254	339	432.7	618.7	643.7

Table 19. Loss Analysis Source Resolution Time Data

Packet Count	End-to-End Loss Rate						
	0.01	0.05	0.1	0.2	0.35	0.5	0.75
1	0.0321	0.0321	0.034	0.0363	0.0377	0.0467	0.0467
2	0.0499	0.0499	0.0523	0.0559	0.0551	0.0680	0.1048
4	0.0871	0.0889	0.0945	0.0961	0.1046	0.1228	0.1652
8	0.1589	0.1589	0.1698	0.2024	0.2342	0.2396	0.3857
16	0.3042	0.3933	0.4130	0.4110	0.4143	0.4997	0.9714
32	0.5950	0.7227	0.7621	0.8125	1.2063	1.2555	2.6179
64	1.2556	1.6827	1.5977	1.9561	2.1577	2.5867	4.5876

Table 20. Loss Analysis Source Packet Count Data

Packet Count	End-to-End Loss Rate						
	0.01	0.05	0.1	0.2	0.35	0.5	0.75
1	11	11	12.3	15	11.7	18	15
2	15	15	15.7	18.3	17.7	22.7	35
4	24	25.3	27	28.3	30	33.7	50.3
8	42	42	45	53.7	63.3	61.3	104.3
16	82	105.7	112	111	111	132.7	248.7
32	162	194	203	216.7	306.7	319	642.7
64	339	432.7	413.7	501	558.3	647	1134.3

Vita

David Chunglin Lee

David C. Lee obtained his Bachelor of Science in Computer Engineering and his Master of Science in Electrical Engineering, both at Virginia Tech. He has been a Graduate Research Assistant with the Southeastern University and College Coalition for Engineering Education (SUCCEED) and is more recently an Instructor for the Bradley Department of Electrical and Computer Engineering (ECE). He has taught courses on advanced internet architectures and microprocessor design. He also coordinates experimental IPv6 service at Virginia Tech and is involved in 6bone deployment work. His professional memberships include IEEE and Eta Kappa Nu. He has also served as a conference reviewer, the IEEE Region 3 Student Representative, the Graduate Student Assembly's representative to the University's Commission on Research and an active volunteer and officer of various other organizations. His research interests include distributed virtual environments/distributed interactive simulations, high performance and active network architectures, intelligent agents, and network protocols.