INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Performance Analysis and Improvement of the Open Shortest Path First Routing Protocol

by

Mahad M. Ahmed, B. Eng.

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario
CANADA, K1S 5B6

September 13, 1999

© 1999, Mahad M. Ahmed

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*

Our file *Notre référence*

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-48446-7

Canada

The undersigned recommend to the Faculty of Graduate Studies
and Research the acceptance of the thesis

Performance Analysis and Improvement of the Open Shortest Path First Routing Protocol

submitted by Mahad M. Ahmed, B.Eng. in partial fulfillment of the
requirements for the degree of Master of Engineering

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

Carleton University
September 13, 1999

Abstract

Open Shortest Path First (OSPF) is a popular routing protocol designed to support routing in TCP/IP networks. In this research, the performance of the OSPF protocol was studied. In particular, the UMD OSPF software, a widely used implementation of the OSPF protocol developed at the University of Maryland, was used as the reference implementation. The goals of this study were to characterize the performance of the OSPF software, identify the performance limiting factors in the implementation, and recommend efficient solutions to improve the performance. The performance issues identified in this research are not unique to the UMD OSPF software, rather they can be found in other vendor's implementations of OSPF. The proposed data structures and algorithms are generic and can be used to improve the performance and scalability of other OSPF implementations. The proposed solutions are currently being used in real OSPF products, which have resulted in significant performance improvement.

Acknowledgements

I would like to express my sincere thanks to my supervisor, Professor Dorina C. Petriu, for her consistent support, guidance, and encouragement throughout this research.

I would like to thank the people who have had a valuable impact on this work, especially Hesham El-Sayed, Anant Jalnapurkar, and Jiri Soukup. I would also like to thank the members of the Software Engineering Analysis Lab in Nortel Networks for their support during this research.

Nortel Networks and the Communications and Information Technology of Ontario (CITO) are gratefully acknowledged for their funding support.

A special gratitude is due to my dear mother and beloved uncle for their unconditional support, encouragement and understanding during the course of my studies.

Table of Contents

Abstract.....	iii
Acknowledgements	iv
Table of Contents	v
List of Tables	vii
List of Figures.....	viii
Table of Acronyms.....	xi
Chapter 1 Introduction	1
1.1 Motivation.....	1
1.2 Thesis Contributions	3
1.3 Thesis Content	6
Chapter 2 Background Literature	7
2.1 Historical Overview of the ARPANET Routing Algorithms	7
2.2 Overview of the OSPF Protocol	8
2.3 Overview of the ISPF Algorithm.....	11
Chapter 3 Software Architecture Recovery.....	15
3.1 How Complex is the OSPF Protocol?.....	16
3.2 A Process for Software Architecture Recovery	17
3.3 Recovered Software Architectural Views.....	21
3.3.1 Software Static View	22
3.3.2 Scenario Analysis (Dynamic view)	26
3.3.3 Data Structures.....	33
Chapter 4 Performance Measurement Framework.....	38
4.1 Network Topology Generator	40
4.2 Topology Transformer	42
4.3 Protocol Simulator	44
4.4 Quantify	45

Chapter 5 OSPF Protocol Performance Analysis	46
5.1 Workload Selection.....	47
5.2 Performance Measurements of Basic OSPF Protocol Packets	48
5.3 Performance Measurements of the Link-State Database Exchange Process	50
5.4 Performance Measurements of the Routing Table Calculation Process	53
5.4.1 Intra-Area Routing Table Calculation (Dijkstra's Algorithm)	54
5.4.2 Inter-Area Routing Table Calculation	59
5.4.3 External Routing Table Calculation.....	60
5.4.4 Detailed Analysis: Identification of the Most Expensive Functions	61
Chapter 6 Solutions for Performance Improvement	67
6.1 Optimization Solutions for the cl_enq Function.....	67
6.2 Optimization Solutions for the findLSA Function	68
Chapter 7 Design and Implementation of the ISPF	
Algorithm.....	71
7.1 Data Structures used in the Design	72
7.1.1 The Link-State Database.....	72
7.1.2 The SPF Tree Data Structure	72
7.1.3 The Topology Change List	73
7.1.4 The Candidate List.....	75
7.1.5 The Routing Table	75
7.2 The ISPF Algorithm Components	75
7.3 Performance Measurements.....	82
Chapter 8 Conclusions.....	87
References	92
Appendix A Detailed Example of OSPF Adjacency Forming	
Process.....	98
Appendix B Optimization Solutions for the findLSA	
Function	100
Appendix C Illustrative Scenarios on the Effects of	
Topological Changes within an OSPF Area	105

List of Tables

Table 4.1	Link type probabilities and cost metrics used in backbone areas	44
Table 4.2	Link type probabilities and cost metrics used in leaf areas	44
Table 5.1	Execution cost of various OSPF packets	48
Table 5.2	Performance data for the cl_enq function	64
Table 5.3	Performance data for the findLSA function	66
Table 7.1	Types of topological changes in an OSPF area	74

List of Figures

Figure 2.1	A sample physical view of an OSPF area	10
Figure 2.2	OSPF topology graph	10
Figure 2.3	A conceptual view of the LSAs exchanged between routers	10
Figure 2.4	The SPF Tree generated by R1	11
Figure 2.5	Routing Table for R1	11
Figure 2.6	The new SPF tree after the cost of the link from R4 to R6 increases... 13	
Figure 2.7	The new SPF tree after the cost of the link from R4 to R5 decreases.. 14	
Figure 3.1	Lines of code used to implement each major protocol as a percentage of the total size	16
Figure 3.2	A process for software architecture recovery.....	17
Figure 3.3	A graphical view of the files in the system and their interconnections	20
Figure 3.4	Architectural Notations	22
Figure 3.5	Static view of software architecture of major components	25
Figure 3.6	Simplified example of the typical interactions between two OSPF routers showing four different scenarios.	27
Figure 3.7	Receiving first Hello packet scenario.....	29
Figure 3.8	Receiving first Database Description packet scenario	30
Figure 3.9	Receiving last Database Description packet scenario	31
Figure 3.10	Receiving Link-State Update packet scenario.....	32
Figure 3.11	Main classes and data structures in the system	33
Figure 3.12	Various data structures used to implement the link-state database	35
Figure 3.13	The AVL tree data structure used to implement the routing table	36
Figure 3.14	The organization of the Next Hop Entry Pool and the Next Hop Entry Hash Table.....	37
Figure 4.1	A framework for software performance measurement.....	39
Figure 4.2	Example of Internet domain structure	41
Figure 5.1	Cost of processing a DD packet as a function of the number of LSA summaries it contains	49

Figure 5.2	Cost of processing a Link-State Update packet as a function of the number of LSAs it contains.....	50
Figure 5.3	The OSPF neighbour finite state machine.....	51
Figure 5.4	The performance of the link-state database exchange activity as a function of the number of LSAs exchanged.....	52
Figure 5.5	Conceptual view of the different routing hierarchies within OSPF.	54
Figure 5.6	The effect of network connectivity on Dijkstra in a backbone area.	57
Figure 5.7	The effect of network connectivity on Dijkstra in a leaf area.	57
Figure 5.8	The effect of number of routers on Dijkstra in a backbone area.	58
Figure 5.9	The effect of increasing number of routers on Dijkstra in a leaf area.	58
Figure 5.10	The cost of inter-area routing table calculation.	59
Figure 5.11	The cost of external routing table calculation.	60
Figure 5.12	The effect of increasing network connectivity on the performance of Dijkstra's algorithm (detailed view).....	62
Figure 5.13	The effect of increasing number of routers on the performance of Dijkstra's algorithm (detailed view).....	63
Figure 5.14	A conceptual view of inserting a new vertex into the candidate list... ..	64
Figure 5.15	A conceptual view of how the findLSA function works.....	65
Figure 6.1	Performance measurements of Dijkstra's algorithm using different data structures to implement the candidate list.	68
Figure 6.2	Performance comparison of the original implementation of Dijkstra's algorithm and an implementation using the caching method.....	69
Figure 6.3	Performance comparison of the original implementation of Dijkstra's algorithm and an implementation with the two proposed solutions.....	70
Figure 7.1	A sample SPF tree	73
Figure 7.2	The pointer-based implementation of the SPF tree	73
Figure 7.3	Two different instances of the LSA generated by router R4.....	76
Figure 7.4	ISPF algorithm implementation components	78
Figure 7.5	High-level flowchart for the ISPF Algorithm	79
Figure 7.6	High-level flowchart for the modified SPF Algorithm	82
Figure 7.7	The total execution cost of the ISPF and SPF in a backbone area.	83

Figure 7.8	The total execution cost of ISPF and SPF algorithms in a leaf area. ...	84
Figure 7.9	Large-scale histogram of ISPF subtree sizes.	85
Figure 7.10	Small-scale histogram of ISPF subtree sizes.	85
Figure 7.11	Detailed comparison between the execution costs of ISPF and SPF algorithms in a backbone area.	86
Figure A.1	An example on the process of forming adjacency between two OSPF routers.	99
Figure B.1	The relationship between the LSDB and LSA data structures	101
Figure B.2	The modifications done to the LSA data structure to facilitate caching of pointers to neighbor nodes.	101
Figure B.3	Physical network view of a simple OSPF area.	102
Figure B.4	The corresponding topology graph for the simple OSPF area shown in Figure B.3.	102
Figure B.5	The interconnections between the various data structures	103
Figure C.1	OSPF LSAs exchanged after the link between R4 and R6 fails, and the changes in the area topology in scenario A1	106
Figure C.2	The topology change list for scenario A1.	107
Figure C.3	The Candidate List for scenario A1.	107
Figure C.4	The SPF tree for scenario A1	107
Figure C.5	The topology change list for scenario A2.	108
Figure C.6	The Candidate List for scenario A2.	108
Figure C.7	OSPF LSAs exchanged after R6 fails, and the changes in the area topology in scenario B1	109
Figure C.8	The topology change list for scenario B1.	109
Figure C.9	The Candidate List for scenario B1.	109
Figure C.10	The SPF tree for scenario B1	110
Figure C.11	The topology change list for scenario B2.	111
Figure C.12	The Candidate List for scenario B2.	111

Table of Acronyms

AS	Autonomous System
AVL	Adelson, Velskii & Landis tree data structure
BGP	Border Gateway Protocol
DD	OSPF Database Description packet
EGP	Exterior Gateway Protocol
IGP	Interior Gateway Protocol
ISPF	Incremental Shortest Path First Algorithm
LSA	Link State Advertisement
LSDB	OSPF Link State Database, also the name of a data structure used to implement the link state database
LSU	OSPF Link State Update packet
NSP	Network Service Provider
OSPF	Open Shortest Path First Routing Protocol
RIP	Routing Information Protocol
RPS	Routing Protocol Simulator
SPF	Dijkstra's Shortest Path First Algorithm

Chapter 1

Introduction

1.1 Motivation

The growth of the global Internet has been phenomenal. Originally built to link a small community of researchers, the Internet is now used by an estimated 150 million users [Petrazzini99]. In January 1999, the number of Internet hosts had tripled from the same time in 1997, to an estimated 43 million hosts [Lottor99].

The traffic that the Internet must carry is also growing at a phenomenal rate. In 1996, a survey of major network service providers (NSPs) revealed that for different providers, traffic loads have more than quadrupled over the same time in the previous year [Bruno96]. The type of traffic on the Internet is changing to include increased use of voice, image, and video. These types of traffic require real-time response and quality of service.

The explosive growth in the size of the Internet and traffic volume has not come without a price - serious performance and scalability problems have plagued NSPs. During at least one period in 1996, some NSPs posted packet loss rates of 30% to 50%, while throughout the year some NSPs experienced losses approaching 30% with regularity. Much of these problems can be traced not just to the sheer volume of traffic but also to the poor performance of routers in maintaining and updating their routing tables. The poor performance of routers can be linked to a phenomenon known as *route flapping* [Bruno96, Stallings97].

Route flapping occurs when a router intermittently ceases transmission. This can be caused by configuration errors, link failures, software bugs and other problems. When

other routers discover that a neighboring router has ceased, they will stop forwarding traffic to the ceased router, and will recalculate their routing tables to reflect this situation. After some period of time, the ceased router might come up again causing other routers to update their routing tables and resume forwarding data to the router. This instability has caused routers to waste a large percentage of their CPU processing resources performing routing table recalculation [Bruno96].

The key in building a high-performance scalable network is the router. The router must be able to quickly update its routing table in face of topological changes in the environment (router/ link failures), be able to pick a route that is appropriate to a given class of traffic and compensate for congestion in the internetwork, and be able to forward IP datagrams at extremely high rates [Stallings97].

Routing protocols must continually update their routing tables in face of topological changes in the network. A failure in the network can change the best paths to certain destinations. It takes some time for any routing protocol to compute the new best paths, and the paths used in the meantime are sometimes sub-optimal or even non-functional. The process of finding the new path after the network changes is called *convergence* [Moy98a]. The preferred routing protocols are the ones that update the routing table quickly and have short convergence time.

Open Shortest Path First (OSPF) is a popular hierarchical routing protocol designed to support routing in TCP/IP networks, and is one of today's leading and most widely used interior gateway routing protocols [Moy98b]. The OSPF protocol is playing an increasingly important role in modern networks, including tactical communication systems that use the Internet Protocol (IP) [Wollman95]. It is more reliable and less bandwidth-intensive than some older protocols like the Routing Information Protocol (RIP) [Moy98a, Hedrick88].

OSPF is not a perfect protocol. It is complex with many configuration options and features [Comer98, Huitema95, Moy98b]. It also requires a great deal of CPU

processing power and memory to keep the link-state database and the routing table up to date [Moy98a, Moy91]. Some of the older routers running RIP might have to be upgraded in order to handle the overheads caused by OSPF. There is a clear need for a detailed and quantitative examination in order to characterize and improve the performance of OSPF routers. However, the public domain literature contains very few studies on this subject (usually companies keep performance information of such kind confidential). The only related research we found was a simulation-based study by Sidhu et al. [Sidhu93], which was limited in scope, and actual measurements did not play a significant role in it.

1.2 Thesis Contributions

This thesis presents the results of characterizing and improving the performance of medium-scale routing software (40K lines of code), which implements the OSPF protocol. In this research, the performance of the UMD OSPF software was studied, a publicly available and widely used implementation of the OSPF protocol. The UMD OSPF software is written in C for the UNIX environment, and was developed at the University of Maryland [Moy98a].

The general objectives of this research were to understand the performance characteristics of the UMD OSPF software, to identify bottlenecks in the implementation, and to propose solutions to alleviate their effects and improve the performance of the system. In this thesis, the term "bottleneck" means the most expensive part of the code, which consumes the most resources (CPU time, memory, etc.).

During the course of this research, we were unable to get design documentation for the system. Also, we were unable to contact the original designers. Since having a good understanding of the software architecture of a system is a necessary step for performance improvement, the source code was reverse engineered and the architecture of the UMD OSPF software was successfully recovered.

Research in algorithm complexity characterizes the behavior of different algorithms at extreme loads, and provides asymptotic results regarding the performance of the algorithm at the extreme loads [Weiss96, Ahuja93]. Therefore, algorithm complexity is an indication of how well the algorithm scales up, but it does not describe in enough detail all the factors contributing to the performance of the implementation at realistic loads (instead of extreme loads) around the operating point of the system. Detailed measurements under different network topologies and workloads are necessary to identify the actual sources of performance bottlenecks and to eliminate them.

In order to characterize and improve the performance of the software, a systematic process for performance measurement and improvement was followed. The first step in this process was to develop an experimental performance measurement framework used to conduct reproducible experiments in a controlled environment under different network topologies and workloads. The role of the framework is to simulate the network around the router under study, in order to provide "realistic" demands from neighboring routers. The next step was to identify long and continuous activities that are frequently executed, and to measure their end-to-end execution costs. Then, to perform more detailed measurements to uncover the most expensive operations within these scenarios. The final step was to optimize these most frequent and expensive components.

The routing table calculation activity, which is implemented using Dijkstra's Shortest Path First (SPF) algorithm [Dijkstra59], was examined in detail. It was discovered that the implementation of Dijkstra's algorithm in the UMD OSPF software has a critical deficiency. The current implementation requires a complete recalculation of the routing table as a result of changes to the OSPF area's topology. In some cases the type of change does not warrant complete recalculation of the routing table, since only a subset of the shortest paths are affected by the topological change. However, the current implementation of the Dijkstra's algorithm is not able to take advantage of these incremental changes.

To overcome this deficiency, a variant of Dijkstra's algorithm called the Incremental SPF (ISPF) algorithm was implemented. This algorithm is much more efficient than the standard Dijkstra's algorithm since it requires only incremental calculations to the routing table in the event of topological changes. This resulted in significant reduction in the cost of updating the routing table.

Summary of Contributions

1. Successfully reverse engineered the UMD OSPF software and recovered its architecture. This was a necessary step in order to understand the software and be able to characterize and enhance its performance.
2. Developed a measurement framework for routers based on existing tools, which is used to measure the performance of the router under realistic workloads. The workloads are produced by a network simulator which models "the rest" of the network, "talks" with the router under study and runs on a different workstation. The framework allows the user to control the network parameters and to perform reproducible experiments.
3. Proposed efficient optimization techniques for the routing table calculation activity (Dijkstra's algorithm) at the data structure level to alleviate the effects of the expensive operations within the algorithm's implementation. The proposed solutions have resulted in significant gains in performance. For example, by using these techniques, 40% performance improvement in the routing table calculation activity (in a backbone area with 400 nodes) was achieved.
4. Demonstrated the successful application of the Incremental Shortest Path First (ISPF) algorithm to improve the performance of the routing table calculation activity in OSPF. The performance experiments show that the ISPF algorithm clearly outperforms the conventional Dijkstra's algorithm in terms of the computational requirements. For example, it was found that the ISPF algorithm was 87% faster than Dijkstra's algorithm to update the routing table in the event of a topological change in the area (for a backbone area with 400 nodes).

The performance issues identified in this research are not unique to UMD OSPF. Three other OSPF implementations (for proprietary reasons, the names of these products can not be revealed) were examined, and similar deficiencies were found. The proposed solutions in this thesis, including the ISPF algorithm, are generic and can be used to improve the performance and scalability of other OSPF implementations. The proposed solutions are currently being used in various Nortel Networks routing products, which have shown significant performance improvement in their products.

The work done during the course of this research have resulted in the following two workshop posters:

- 1- Ahmed, Mahad, "*Performance Analysis of a Routing Protocol Software*", a poster presented at the second IEEE Ottawa Network Operations and Management Systems (NOMS) workshop held at Montebello, Quebec, June 1999.
- 2- Ahmed, Mahad. "*Building Scalable High-performance OSPF Software*", a poster presented at the CITO Research Forum held at Carleton University, Ottawa, May 1999.

1.3 Thesis Content

This thesis is organized as follows. Chapter 2 provides background information on the ARPANET routing protocols, the OSPF protocol, and the ISPF algorithm; Chapter 3 presents the recovered architecture of the UMD OSPF software; Chapter 4 describes the performance measurement framework; Chapter 5 presents performance analysis for the OSPF protocol; Chapter 6 describes two proposed solutions to improve the performance of the routing table calculation activity; Chapter 7 presents the design of the ISPF algorithm and presents performance measurements comparing the SPF and ISPF algorithms, and finally Chapter 8 provides the concluding remarks.

Chapter 2

Background Literature

2.1 Historical Overview of the ARPANET Routing Algorithms

The routing protocols used in most of today's computer networks can be classified as distance-vector or link-state routing protocols [Huitema95, Moy98a]. A distance-vector routing protocol requires that each router simply inform its neighbors of its routing table. For each destination, the receiving router picks the neighbor advertising the lowest cost to that destination, then adds this entry into its routing table for re-advertisement. RIP is a well-known example of distance-vector routing protocols [Hedrick88]. On the other hand, a link-state routing protocol requires each router to maintain an identical map of the entire area. Using this map, each router runs a specialized algorithm to determine the best path to each reachable destination. When a topological change occurs (i.e., a router or a communication link is down), a notification message, called a Link State Advertisement (LSA), is flooded throughout the area. All the routers will update their topological maps, and recompute their routing tables accordingly. OSPF is a link-state routing protocol [Moy98b].

The original routing algorithm of the ARPANET packet switching network, first implemented in 1969, was based on the distance-vector technology [McQuillan78]. In this algorithm, neighboring routers exchange their tables of estimated shortest distance (estimated delay) to each destination every 128 ms. Once the routing tables are received from all adjacent routers, a router could easily determine which adjacent router would result in the shortest delay to each destination in the network. [McQuillan80]

The original routing algorithm of the ARPANET had some fundamental deficiencies. The algorithm's deficiencies included slow response to topological changes, the transmission of large packets containing the routing tables that interfered with data traffic, and a tendency to form routing loops that would persist for seconds at times. The reason behind forming routing loops is that each router based its routing calculation on its local information together with calculations made at every other router. With such a distributed routing calculation scheme, it is difficult to ensure that routes used by different routers are consistent, especially in the event of topological failures. In summary, the algorithm was taking a long time to converge and was consuming considerable network bandwidth in the process [Bertsekas87, McQuillan78, Moy98a].

These fundamental deficiencies were finally corrected in 1979 when the algorithm was replaced by a newer version. The new routing algorithm was the first ever link-state routing protocol and it has formed the starting point for all other link-state protocols, including OSPF [Bertsekas87, Moy98a].

2.2 Overview of the OSPF Protocol

The Open Shortest Path First (OSPF) protocol is a dynamic, hierarchical routing protocol designed to support routing in TCP/IP networks, developed by the OSPF Working Group of the Internet Engineering Task Force [Moy98b]. OSPF is the preferred Interior Gateway Protocol to be used inside large TCP/IP routing domains. As its acronym indicates, OSPF has two main characteristics. The first characteristic is that it is an open protocol, since its specification is in the public domain. The second characteristic is that it uses Dijkstra's Shortest Path First algorithm [Dijkstra59] to calculate the routing table.

High-level architectural decisions for network design were based on the complexity of the shortest-path algorithms involved and the need to improve their scalability.

Hierarchical routing is one of the architectural techniques commonly used when building large scalable networks [Zhu98]. In a hierarchical routing protocol, like OSPF, a routing domain is partitioned into regions, or areas. Within an area, all routers know about the area's complete topology. However, the details of any particular area are hidden from all other areas.

The hierarchical routing technique can be viewed as decomposing a large graph into a set of smaller sub-graphs, and then computing the shortest paths within each sub-graph separately. Thus, the extent of the shortest-path algorithm and its complexity is limited to a single sub-graph (area) at a time. This routing scheme keeps the routing table sizes, and therefore the memory and CPU demands on the routers to a manageable level [Moy98a].

IP routing protocols can be divided into two classes: Interior Gateway Protocols (IGPs) and Exterior Gateway Protocols (EGPs) [Comer98, Huitema95]. IGPs are used for routing networks that are under a common administration (i.e., an Autonomous System). EGPs are used to exchange routing information between networks that do not share a common administration. OSPF is an Internal Gateway Protocol that is designed to be used inside an Autonomous System (AS).

Each participating router inside an OSPF area maintains an identical topological database, called the link-state database. A link-state database is essentially an overall picture of the networks and routers inside the area, and their interconnections. Each individual piece of this database, called a Link-State Advertisement (LSA), describes a particular router's local state (i.e., the state of the router's interfaces and its reachable neighbors). Figure 2.1 shows a sample OSPF area with 7 routers. Figure 2.2 shows the topological graph assembled by each router from its link-state database. Finally, Figure 2.3 shows an example of the LSAs generated by routers R4, R5, R6 and R7.

Figure 2.1: A sample physical view of an OSPF area

Figure 2.2: OSPF topology graph

LSA from R4		LSA from R5		LSA from R6	
Source Router R4		Source Router R5		Source Router R6	
Link From R4 to	Cost	Link From R5 to	Cost	Link From R6 to	Cost
R2	1	R3	5	R4	1
R5	5	R4	5	R5	3
R6	1	R6	3	R7	3
		R7	1		
		LSA from R7			
		Source Router R7			
		Link From R7 to	Cost		
		R5	1		
		R6	3		

Figure 2.3: A conceptual view of the LSAs exchanged between some of the routers in the OSPF area

From the link-state database, each router builds the routing table by calculating the shortest paths to all possible destinations using Dijkstra's algorithm. Dijkstra's algorithm maintains a list, called the *candidate list*, to contain all the vertices to examine in the next iteration. The reason behind the name "candidate list" is that the list contains the nodes that are potential candidates to be added to the shortest-path tree in the next iteration of the algorithm. The algorithm starts by adding the source (or the calculating) router to the shortest-path tree. All of the source router's neighbors are then added to the candidate list, with costs equal to the cost of the links from the source router to its neighbors. The router on the candidate list with the

smallest distance from the source router is removed from the list and added to the shortest-path tree. Then, the neighbors of this router are examined for possible inclusion in (or modification of) the candidate list. The algorithm terminates when the candidate list becomes empty.

2.3 Overview of the ISPF Algorithm

The ARPNET routing algorithm had three components: a link delay calculation procedure, a packet flooding process, and the central component was the routing table calculation algorithm. Instead of using the conventional Dijkstra's Shortest Path First Algorithm (SPF) algorithm to perform the full routing table calculation whenever a topological change occurs, McQuillan et. al. developed a more efficient variant of the SPF algorithm, called Incremental SPF (ISPF) [McQuillan80]. By using ISPF, changes in area topology require only an incremental calculation rather than a complete recalculation of all shortest paths (the routing table).

Before examining the ISPF algorithm in detail, let us look at an example, which demonstrate in a concrete way why the standard SPF algorithm is inefficient. Figures 2.4 and 2.5 show the SPF tree and routing table generated by router R1 for the OSPF area shown in Figure 2.1.

Destination	Path Cost	Next Hop
R2	1	R2
R3	1	R3
R4	2	R2
R5	6	R3
R6	3	R2
R7	6	R2

Figure 2.4: The SPF Tree generated by R1 Figure 2.5: Routing Table for R1

Consider the case where the communication link between R4 and R5 goes down. Clearly this link is not in the SPF tree and the fact of it being down should not change the routing table, since it was an expensive link anyway. However, the standard SPF

algorithm requires complete routing table calculation to be done whenever a topological change happens in the OSPF area irrespective of the relevance or the impact of the change. This unnecessary complete recalculation of the routing table leads to a waste of computational resources.

The ISPF algorithm is best explained using an example. Let us consider the impact of the following topological changes, in the OSPF area shown in Figure 2.1, on the SPF tree. In the following cases, assume that S is the node (router) calculating the routing table, and it is the root of the SPF tree.

First, consider the general case where the cost of the communication link from node (router) A to node (router) B increases. There are two possibilities regarding the impact of the change:

- In the first possibility, the link is not in the SPF tree, meaning that the shortest path from S to node B does not pass through node A. Then, no changes to the SPF tree are required, since the link was not part of any shortest path prior to the change, and it will certainly not be used when its cost increases.
- In the second possibility, the link is in the SPF tree. In this case the cost to reach node B will increase, as does the cost to reach each node whose route from S passes through B. Thus, nodes in the subtree rooted at B are candidates for changed positions in the SPF tree. Conversely, nodes not in this subtree will not be repositioned. The ISPF algorithm handles an increase of X units in the link cost from A to B as follows. First isolate all the nodes in the subtree rooted at B and increase their path costs from S by X units. Then for each node N of the subtree, try to find a shorter path to N via an adjacent node which is not already in the subtree. If such a path is found, then add N to the Candidate List. At the end of these steps, the Candidate List will either be empty or will contain some subtree nodes for which better paths have been found. If the list is empty, the ISPF algorithm terminates. Else, Dijkstra's SPF algorithm needs to be invoked

using the populated Candidate List and the previous SPF tree as an input, in order to find the best possible shortest paths for the nodes on the list and possibly other subtree nodes. Figure 2.6 shows the modifications to the SPF tree of Figure 2.4 that result when the cost of the communication link from router R4 to router R6 increases from 1 to 8.

Figure 2.6: The new SPF tree after the cost of the link from R4 to R6 increases

Now, consider the case where the cost of the communication link from node A to node B decreases by X units. There are two possibilities regarding the impact of the change:

- In the first possibility, the link is in the SPF tree. In this case paths to nodes in the subtree rooted at B will be unchanged, since these are already the shortest paths, but the costs of these paths will be shortened by X units. In addition, any node whose distance from S is less than or equal to B's new distance from S will not be repositioned, since the path to the node must reach B first in order to take advantage of the improved link cost. However, nodes which are not in the subtree and which are farther from S than B may have a shorter distance via one of the subtree nodes. The ISPF algorithm handles this case by first isolating all the nodes in the subtree rooted at B and decreasing their path costs from S by X units. Then for each node N that is not in the subtree but is adjacent to a subtree node, try to find a shorter path to N via an adjacent node which is in the subtree. If such a path is found, then add N to the Candidate List. At the end of these steps, the Candidate List will either be empty or will contain some non-subtree nodes for

which better paths have been found. If the list is empty, then the ISPF algorithm terminates. Else, Dijkstra's SPF algorithm needs to be invoked using the populated Candidate List and the previous SPF tree as an input, in order to find the best possible shortest paths for the nodes on the list and possibly other non-subtree nodes.

- In the second possibility, the improved link is not in the SPF tree. If the cost of the path from S to B using the improved link is greater or equal than the cost of the original shortest path to B, then B can not take advantage of this improvement, and there will be no changes to the SPF tree. If, on the other hand, the new path is shorter than the original path, then the best route to reach B will pass through A. The first change to the SPF tree is, therefore, to relocate B (and its subtree), attaching it to node A via the improved link. Now the situation is identical to that of the first possibility in which the link from A to B was in the SPF tree in the first place and its cost decreased. Figure 2.7 shows the modifications to the SPF tree of Figure 2.4 that result when the cost of the communication link from router R4 to router R5 decreases from 5 to 1.

Figure 2.7: The new SPF tree after the cost of the link from R4 to R5 decreases

Finally, there are other types of topological changes that can happen in addition to the previous changes that were discussed, such as the entire failure (or recovery) of a node or link. The ISPF algorithm handles these cases in a similar fashion by isolating the affected nodes by the topological change and recalculating the best paths to these nodes. For a more formal description of the ISPF algorithm, the reader is referred to [McQuillan80].

Chapter 3

Software Architecture Recovery

The software architecture defines how a system is designed and how the various components are connected to, and operate with, each other. Recovering the architecture is a necessary step for re-documentation, design recovery, architectural analysis for the purposes of maintenance or re-engineering, and performance improvement.

The task of analyzing and reasoning about the architecture becomes more challenging when the architectural documentation is out of sync with the implemented system or does not exist, as was the case with the UMD OSPF software. As a consequence, designers need to be able to extract information from an existing system implementation and reason architecturally about this information.

This chapter describes the reverse engineering efforts performed on the UMD OSPF system in order to recover its architecture. The following are the products of the reverse engineering efforts:

1. Capturing the static view of the system, which describes its major components and their interconnections.
2. Capturing the dynamic view of the system, which shows its behavioral aspects. The dynamic view is obtained by tracing the execution of commonly used scenarios.
3. Identifying important data structures and capturing their relationships.

This chapter is organized as follows. Section 3.1 describes the complexity of the OSPF protocol. Section 3.2 describes the process used to recover the software architecture of the system. Section 3.3 presents the recovered architectural views,

which include the static view, the dynamic view, and the important data structures used by the OSPF system.

3.1 How complex is the OSPF Protocol?

The OSPF system used in this study is written in C++ and C, and it is composed of 30 files or about 40,000 lines of code. A good measure of how difficult the reverse engineering task was is the complexity of the system involved. OSPF is a very complex protocol with many configuration options and features [Comer98, Huitema95, Moy98b]. The latest OSPF specification [Moy98b] contains 245 pages, when less than 100 pages are used to describe RIP [Hedrick88].

One way to evaluate the complexity of the OSPF protocol is to compare the number of lines required to implement it with the number of lines required to implement other protocols. Even though the number of lines of code depends on the coding style, it is a measure of the relative complexity of the protocol. Comer [Comer98] compares nine major protocols, including OSPF, in terms of the number of lines needed to implement each protocol as a percentage of the total number of lines. Figure 3.1 shows that the size of the OSPF implementation is slightly larger than IP, and three and one-half times the code required for RIP.

Figure 3.1: Lines of code used to implement each major protocol as a percentage of the total size [Comer98]

3.2 A Process for Software Architecture Recovery

To recover the software architecture of the system, a systematic process for architecture recovery was followed. The building blocks of the architecture recovery process are shown in Figure 3.2.

Figure 3.2: A process for software architecture recovery

A detailed explanation of each activity in the architecture recovery process is given below:

Step 1: Code Analysis

The recovery of the software architecture cannot be done manually when dealing with a medium-scale system, like the UMD OSPF software. Reverse engineering tools are needed to analyze the software and extract source code information. We experimented with the following two reverse engineering tools to aid us with this task:

1. ***SNiFF+*** [SNiFF+], which is an integrated development environment for C and C++ developed by TakeFive software. We found that ***SNiFF+*** works well for analyzing and understanding small systems, however it simply cannot process, classify and visualize architectural data for medium or large-scale systems like the UMD OSPF software.
2. ***Source Navigator*** [Source Navigator], which is a tool developed by Cygnus Solutions that analyzes the source code and creates a database that captures the interrelationships between all entities in the source code. We found that Source Navigator is much faster and capable of handling larger-sized systems than ***SNiFF+***.

We decided to use ***Source Navigator*** to analyze the software and build a project (cross-reference) database. The cross-reference database contains information about entities (constants, local variables, macros, structures, functions, methods,...) referenced in the system. For each entity, the database stores the name of the source file in which the entity is defined or implemented, and the list of entities that reference it.

We noticed that both ***Source Navigator*** and ***SNiFF+*** had difficulties with function pointers. Neither of the two tools recognized that the de-reference of a function pointer was a function call. This deficiency was remedied by augmenting these tools with the use of a profiler/debugger.

Step 2: Filtering and Parsing

A simple Perl script was developed by the author to parse and filter the cross-reference database to identify method (or function) dependencies and to identify the interconnections between the source files in the system. Two files are defined to be interconnected if a method in one file references a method in the other file.

Step 3: Clustering Process

The key concept of the clustering process is to group similar entities into clusters, such that intra-cluster similarity or cohesion is high, and inter-cluster similarity or coupling is low [Lung98a]. The clustering process combines semantically related components (files) into more abstract objects like subsystems to provide a high-level topological view of the system, and allows designers to relate the code to application domain concepts. Each cluster obtained corresponds to a particular domain concept and is responsible for a set of cohesive responsibilities.

The clustering method we employed is based on component interconnection as described in [Lung98a]. Lung provides empirical evidence that clustering based on interconnectivity of entities is practical and useful [Lung98a].

In order to apply the clustering technique, the following are the steps involved:

1. First, the coupling information (the interconnections) among different files in the system is captured using the cross-reference database. Figure 3.3 shows a graphical view of the files in the system and their interconnections
2. Then, an automated clustering tool developed by Nortel Networks (for proprietary reasons, the name of the tool can not be revealed) was used to group related files into logical clusters such that intra-cluster similarity (cohesion) is high and inter-cluster similarity (coupling) is low. The output of the clustering tool was a set of logical clusters. This tool was selected because it was the only automated clustering tool available at the time of this research.

Figure 3.3: A graphical view of the files in the system and their interconnections

Step 4: Analysis

Since reverse engineering tools alone are not enough to produce architectural abstractions, human intervention and interpretation of tool-extracted artifacts is crucial. At the analysis stage, domain knowledge was used to validate and refine the logical clusters obtained from Step 3 by relating and matching the resulting clusters with domain concepts. Then, using method-coupling information, obtained from Step 2, the static view of the system was produced.

Related Work

Murphy et al. [Murphy97] have developed a process for architecture capture and validation, which lets designers validate their high-level reasoning with information from the source code. In this process, the designer defines a high-level model of interest, extracts a source model (such as a call graph) from the source code, and defines a declarative mapping between the two models. A software reflexion model is then computed to determine where the high-level model does and does not agree with

the source model. The designer interprets the reflexion model and, as necessary, modifies the input to iteratively compute additional reflexion models.

Murphy's approach differs in a number of ways from the approach followed in this research. First, in Murphy's approach, the designer must specify the high-level entities explicitly, whereas the architectural recovery process presented in this chapter infers them instead. Second, her approach focuses on comparing high-level and source models, rather than on discovering high-level architecture.

A different file-clustering scheme is used by Anquetil et al. [Anquetil98]. Instead of clustering files based on their interconnections, they cluster files based on their names. The basic idea behind this approach is that file names refer to concepts, and files with common concepts can be combined into a single cluster. This approach works to some extent and may be useful for some specific systems. However, there are some practical problems with file names based approaches. The accuracy of these approaches is highly dependent on the consistency of the naming conventions employed within the system. Moreover, this approach may not be applicable to joint venture projects, since different file naming conventions can be employed across organizational boundaries [Lung98a].

3.3 Recovered Software Architectural Views

A combination of multiple views is often necessary for the process of architectural reconstruction. It has been generally accepted that a complete view of a software architecture requires different views to be fused [Kazman98a]. Fusing architectural views means establishing connections between them. This is important for several reasons. First, different views provide complementary information. Second, users need to be able to navigate among views to completely understand the architecture.

Information extracted from one view can be improved with information from another view. For example, a parser and a profiler both produce information concerning functions and calls between them, but the former identifies potential calls (static

information) while the later identifies actual calls. With the use of late binding in programming languages, important architectural relations are not bound at compile time, and these relations can only be discovered during run-time. Therefore, a source view alone is insufficient, and a dynamic view alone is insufficient. In the following subsections, three views of the OSPF system are presented: the static view, the dynamic view, and the data structures view.

3.3.1 Software Static View

The static view of the system reveals how it is constructed from smaller interconnected functional components (or clusters). The simple lexicon shown in Figure 3.4 is used to portray the structure of the system.

Figure 3.4: Architectural Notations

In the figure, square-cornered boxes with thick solid lines represent C++ objects, round-cornered shaded boxes represent logical software components (clusters of closely related files), small round-cornered rectangles represent methods or functions, solid black arrows represent a method or function invocations, and dashed black arrows represent actual method or function invocations during run-time. In addition, to improve the readability of the views, a small symbol, called *Port*, is introduced to reduce the number of connections (arrows) between clusters (this symbol was adapted

from [Selic94]). This simple lexicon provides a reasonable static and dynamic representation of the architecture.

The static view of the system is shown in Figure 3.5, which depicts the overall topology of the system by capturing the system components and their interconnections. A brief description of the logical components (clusters) identified in the system is given below:

1. **OSPF Packet Reception:** responsible for receiving OSPF protocol packets (Hello, Database Description, Link State Request, Link State Update, and Link State Acknowledgment) from the IP layer and processing them. In order for the packet to be accepted by the OSPF protocol, it must pass a number of tests. The tests include checking the validity of the OSPF packet header. Packets that do not pass the tests are discarded. If the packet is accepted, then it should be further processed depending upon its type.
2. **OSPF Packet Transmission:** responsible for implementing the logic needed to transmit the different types of OSPF packets. This includes adding the standard OSPF packet header, and sending the packet through the IP layer for transmission.
3. **LSA Flooding:** responsible for implementing the OSPF flooding procedure used to distribute and synchronize the link-state database with neighbor OSPF routers. It provides the mechanism for flooding components of the link-state database (LSAs) through all eligible outgoing interfaces.
4. **Link-State Database:** responsible for maintaining and manipulating individual pieces of the link-state database. Also implements the database insertion and lookup (access) functions.

5. **State Management:** responsible for managing the state machines of the active interfaces and neighbors of the router.
6. **SPF Algorithm:** responsible for calculating the intra-area, inter-area, and external routes and updating the routing table. The intra-area routes are calculated, for each attached area, by building the shortest-path tree. The inter-area routes are calculated through the examination of summary-LSAs. Routes to external destinations are calculated through the examination of AS-external-LSAs.
7. **Routing Table:** responsible for implementing the functions used to update the routing table AVL tree and the next hop entry block. Also, it includes functions to notify the IP Forwarding Table manager with any changes that happen in the routing table.
8. **Routing Table Manager:** responsible for managing routing information from various routing protocols such as OSPF, BGP, RIP and static routes. Also responsible for the initialization and provisioning of the entire routing system.

Figure 3.5: Static view of software architecture of major components

3.3.2 Scenario Analysis (Dynamic view)

This section describes the execution behavior of four typical scenarios in the OSPF software. These scenarios describe the various stages involved in building adjacency and link-state database exchange between two OSPF routers (for a detailed example on adjacency forming, refer to Appendix A).

To understand what happens in the system during these scenarios, we traced the execution behavior of each scenario and superimposed that behavior on the static view of the system presented in Section 3.3.1. The detailed execution traces of the scenarios are shown in Figure 3.7 through Figure 3.10. In every scenario, the main methods/functions that took place during the execution of that scenario are highlighted. Numbers that appear beside the arrows specify the ordering of method/function invocation.

Link-State Database Exchange

The Database Exchange Protocol is used to synchronize the link-state databases between two adjacent OSPF routers, when they first start communicating. At the beginning of the Database Exchange Protocol, the two routers form a master/slave relationship. Then, each router describes its database by sending a sequence of Database Description (DD) packets to its neighbor; each Database Description packet describes a set of LSAs belonging to the router's database. If the two databases are not fully synchronized, the two routers will exchange link-state requests, updates, and acknowledgements to ensure that their databases are synchronized with the most current information about the area's topology. The various scenarios involved in a link-state database exchange between two OSPF routers (RT1 and RT2) are shown in Figure 3.6. In this configuration, it is assumed that RT1 (router-under-test) has a smaller router ID than RT2.

Figure 3.6: Simplified example of the typical interactions between two OSPF routers showing four different scenarios

As shown in Figure 3.6, once the interface of the router-under-test (RT1) becomes operational, it sends a Hello packet on that interface. The following is a summary of what happens next:

Scenario 1: RT1 receives the first Hello packet from RT2

RT2 responds with a Hello packet, indicating itself as the Designated Router (DR) and that it has heard the Hello packet from RT1. This in turn causes RT1 to go to state ExStart, where it starts to bring up the adjacency. RT1 initiates the Database Exchange Protocol by sending a Database Description (DD) packet, and asserting itself as the master. A detailed execution trace of this scenario is shown in Figure 3.7.

Scenario 2: RT1 receives the first DD packet from RT2

RT2 responds with a DD packet, asserting itself as the master, because it has a higher router ID. This in turn causes RT1 to become the slave, go to state Exchange, use RT2's DD sequence number, and reply back with a DD packet. A detailed execution trace of this scenario is shown in Figure 3.8.

Scenario 3: RT1 receives the terminating DD packet from RT2

RT2 sends an empty DD packet. This DD packet is used to indicate the completion of Database Exchange protocol. When RT1 receives this empty DD packet, it goes immediately to Full state and sends its last DD packet. A detailed execution trace of this scenario is shown in Figure 3.9.

Scenario 4: RT1 receives a LS Update packet from RT2

RT2 sends a Link-State Update packet containing a new LSA. This in turn causes RT1 to install the received LSA in its link-state database. A detailed execution trace of this scenario is shown in Figure 3.10.

Routing Plane

Figure 3.7: Receiving first Hello packet scenario

Figure 3.8: Receiving first Database Description packet scenario

3.3.3 Data Structures

Overview

Understanding important data structures used in a software system and their relationships is an integral part of any reverse engineering or performance analysis effort. Figure 3.11 shows a high level conceptual view of the important classes and data structures used in the routing software.

Figure 3.11: Main classes and data structures in the system

The following is a brief description of these classes and data structures:

- The **IP RTM** class implements the Routing Table Manager. There is one instance of the IP RTM class for each virtual router. Each instance of IP RTM contains only one instance of **IP_INSTANCE** (IPI). The IPI data structure contains pointers to all the important data structures of IP RTM, including the various routing protocols (i.e., OSPF), the routing table, and the next hop entry table.

Each instance of IP RTM contains a reference to **IP Forwarding Table** object, which maintains the best routes that the router will use for forwarding data packets.

- The **OSPF** data structure contains all the information pertaining to the OSPF protocol within a virtual router, such as the configured parameters and cumulative statistics for the entire OSPF protocol. The OSPF data structure also maintains a linked list of all areas that the router is actively participating in.
- The **Area** data structure describes the configured parameters and cumulative statistics of one of the router's attached areas. Each area maintains its own link-state database.
- The **Interface (INTF)** data structure contains information describing the configured parameters and cumulative statistics of one of the router's interfaces inside an area.
- The **Neighbor (NBR)** data structure contains information regarding a single neighbor in the locality of the router. The information includes configured parameters and cumulative statistics of this neighbor. The neighbor data structure also contains all information pertinent to the formed adjacency with the neighbor.
- The **Link-State Database** contains the collection of all link-state advertisements in an OSPF area, which gives a complete description of the topology (the routers and network segments and how they are interconnected). Link-state Databases are exchanged between neighboring routers soon after the routers have discovered each other; after that the databases remain synchronized through a procedure called reliable flooding.

Detailed Description

This section examines the details of important data structures in the system: the link-state database, the routing table, and the next hop entry table.

Link-State Database Data Structures

A router has a separate link-state database for every area to which it belongs. All routers belonging to the same area have identical link-state databases for that area. The area link-state database contains several type of LSAs such as: the router-LSAs, network-LSAs, summary-LSAs which are originated by the area's border routers, and external routes (AS-external-LSAs) which appear only in non-stub areas. Figure 3.12 shows the details of the main data structures that implement the link-state database of an OSPF area.

Figure 3.12: Various data structures used to implement the link-state database

The database is divided into partitions (collections) of LSAs according to their type. Each collection of LSAs is accessed using a hash table (LSDB hashtable), where LSAs with the same hash key are stored in the same linked list in a sorted order. The AREA structure contains an array of pointers to LSDB_HEAD structures. Each LSDB_HEAD structure points to the hash table of a particular collection of LSDBs. Each LSDB data structure contains a pointer to the LSA data structure, which is used to describe the local state of a router or a network.

Routing Table Structures

The routing table is derived from the link-state database. It contains all possible routes calculated by all active routing protocols and the provisioned static routes. It is implemented as an AVL tree of route heads (RT_HEAD structures) as shown in Figure 3.13.

Figure 3.13: The AVL tree data structure used to implement the routing table

Entries in the AVL tree are indexed by a destination IP address. Each RT_HEAD corresponds to a unique IP destination address and mask pair and heads a linked list of route entries (RT_ENTRY structures). Each RT_ENTRY represents a valid route to the RT_HEAD's unique destination and keeps a list of equal cost next hops for that route (same protocol, preference, metric).

The list of RT_ENTRY structures being kept by any RT_HEAD is sorted by preference, metric and protocol order. The first RT_ENTRY is the most preferred. Each RT_ENTRY has a maximum of 3 possible next hops to use in forwarding packets to the destination.

A Next Hop Entry Pool (an array of NH_BLOCK structures) is used to store the list of next hops for all the known routes in the routing table. Since more than one route can have the same next hop, the Next Hop Entry Pool is stored separately from the routing table to save memory. In addition, there is a Next Hop Entry Hash Table used to quickly access next hops in the Next Hop Entry Pool by using (hashing) a known IP address as shown in Figure 3.14.

Figure 3.14: The organization of the Next Hop Entry Pool and the Next Hop Entry Hash Table

Chapter 4

Performance Measurement Framework

The performance of routing protocol software depends strongly on the network topology. In order to carry out a representative and meaningful performance characterization study on the OSPF routing protocol, the study has to be performed in a typical network environment. Unfortunately, setting up experiments and collecting measurements in a real network is very difficult and costly, as it requires real equipment (routers), instruments, and time. In addition, measurements performed in a real network are very difficult to reproduce.

The need for reproducible experiments in a controlled environment with different network topologies and various degrees of connectivity was the motivation behind the development of a reusable performance measurement framework. The framework is composed of independent and low-cost COTS (commercial off-the-shelf) components. The role of the framework is to simulate the network around the router-under-study, in order to provide "realistic" demands from neighboring routers.

The experiments were conducted in a simulated network environment using the performance measurement framework. In the simulated network environment, the router running the UMD OSPF software behaves as if it is connected to a real OSPF network. Using this simulated network environment, the router-under-test is subjected to synthetic workloads, whose characteristics are similar to those of the real workloads and can be applied repeatedly in a controlled manner.

Figure 4.1 shows the performance measurement framework. The framework is composed of the following components:

1. Network Topology Generator (TIERS)
2. Topology Transformer (TT)
3. OSPF Routing Protocol Simulator (RPS)
4. Router Under Test (RUT)
5. Code Profiling Tool (QUANTIFY)

The purpose of the Network Topology Generator is to generate example graphs with similar topological structures as today's large networks. The function of the Topology Transformer is to convert the generated graphs into OSPF network topologies in a format that is compatible with the OSPF Routing Protocol Simulator (RPS). The purpose of RPS is to simulate the behavior of these networks to the Router Under Test (RUT). The profiling tool is used to collect and generate detailed profile information about program execution of the RUT. Finally, the reports generated by the profiler are examined for analysis.

Figure 4.1: A framework for software performance measurement

The following subsections describe the various components of the software performance measurement framework.

4.1 Network Topology Generator

The network model plays an important role in assessing routing algorithms. Previous studies have shown that the performance of routing algorithms is greatly influenced by the methods used to model the network [Calvert97, Doar96, Zegura97]. Thus, realistic network topology models are crucial for quantitative evaluation of routing algorithms and protocols.

In the performance measurement framework, the Tiers method (developed by M. B. Doar) [Calvert97] was used to generate topological structures that reflect the hierarchical domain structure which is present in today's large networks. The Tiers method was used for two reasons. First, the models generated by this method bear more resemblance to real networks than the ones generated manually, or using the conventional random network generators. Secondly, the source code is freely available [Tiers97].

In Tiers, the topological structure of a network is modeled using a graph, with nodes representing switches or routers, and edges representing direct connections (transmission links or networks) between the routers. Two levels of hierarchy are modeled in Tiers, corresponding to transit domains, and stub domains. A stub domain carries only traffic that originates or terminates in the domain. Transit domains do not have this restriction. The purpose of transit domains is to interconnect stub domains efficiently; without them every pair of stub domains would need to be directly connected to each other.

Figure 4.2: Example of Internet domain structure

An example of Internet domain structure generated by Tiers is shown in Figure 4.2. Stub domains generally correspond to campus networks or other collections of interconnected local-area networks (LANs), while transit domains are almost always wide-area or metropolitan-area networks (WANs or MANs).

Two sets of parameters are used to control the coarse properties of the networks generated. The first set of parameters governs the relative sizes of the two levels in the hierarchy:

- T , the total number of transit domains, and N_t , the number of nodes per transit domain.
- S , the number of stub domains per transit domain, and N_s , the number of nodes per stub domain.

The second set of parameters governs the connectivity within a domain (intranetwork connectivity) and the connectivity between domains (internetwork connectivity):

- E_t , the average number of edges (transmission lines or networks) from a node to other nodes in the same transit domain.
- E_s , the average number of edges from a node to other nodes in the same stub domain.
- E_{tt} , the average number of edges from a transit domain to another transit domain.
- E_{st} , the average number of edges from a stub domain to a transit domain.

The Tiers method builds the graph in a piecewise fashion, where the pieces correspond to domains at different hierarchical levels. The method first creates a WAN by randomly placing nodes within a grid. Then nodes with a minimum spanning tree are connected. This process is repeated to create LANs using smaller grid sizes. LANs are connected to the WAN by randomly choosing a node from the LAN and another node from the WAN. Redundant links between the LAN and WAN are created by choosing another random node from the LAN and the nearest node in the WAN.

4.2 Topology Transformer

The topology transformer is a PERL script (developed by the author) which is used to transform the network graph model generated by TIERS into an OSPF network topology model and save it in the format required by the OSPF Routing Protocol Simulator (RPS).

The graph model is converted to an OSPF network model as follows:

- 1- Transit domains are considered (interpreted) as backbone areas.
- 2- Stub domains are considered as leaf areas.
- 3- Nodes will represent routers. Each router is assigned a unique router id.
- 4- Edges will represent networks. Each network is assigned a unique IP address, network mask and cost metric.

- 5- Each router will be linked to its attached networks through interfaces. Every interface is assigned an IP address that is derived from the network address to which it is attached.

Assigning costs to links is a difficult task. The OSPF standard [Moy98b] does not specify how the costs should be assigned to links. The only rule for cost assignment is that it must make sense that the sum of the link costs along the path to a destination represents the total cost of the path. In real life, many vendors provide default settings for the link cost, based upon delay or bandwidth of the link.

In this study, costs are assigned to links in two steps:

1. Link types (bandwidth) are chosen according to some probability scheme. A link inside a backbone area should be high speed. In a Leaf area, there is no restriction on link types.
2. Link costs are calculated from link bandwidth using the following formula [Weeks98]:

$$\text{Link cost} = \text{scaling factor} * \frac{100,000,000}{\text{Bandwidth}(bps)}$$

In the above formula, link cost is set to 100,000,000/Bandwidth of the interface, i.e., the number of seconds needed to transmit 100 million bits over the interface. Link costs are supposed to be integers and the lowest values corresponding to the best routes. The scaling factor is used to differentiate between link speeds above 100 Mbps. This is a common practice among network designers [Weeks98, Thomas98].

Link types used in this experiment and their probability of occurrence, as well as the associated cost metric in both backbone and leaf areas were obtained from [Weeks98, Thomas98, OSPFGroup], and are listed in Tables 4.1 and 4.2.

Link type	Bandwidth	Cost	Prob. of occurrence
LAN segment			
LAN10BaseT	10 Mbps	1000	0.05
LAN100BaseT	100 Mbps	100	0.2
LAN1000BaseT	1000 Mbps	10	0.15
Serials			
T1	1.5 Mbps	6500	0.15
E1	2 Mbps	5000	0.25
Optical links			
OC3	155 Mbps	65	0.1
OC12	622 Mbps	16	0.1

Table 4.1: Link type probabilities and cost metrics used in backbone areas

Link type	Bandwidth	Cost	Prob. of occurrence
WAN			
WAN64K	64 Kbps	156200	0.3
WAN128K	128 Kbps	78100	0.18
WAN2Mb	2 Mbps	5000	0.12
LAN segment			
LAN10BaseT	10 Mbps	1000	0.28
LAN100BaseT	100 Mbps	100	0.08
LAN-TokenRing	4 Mbps	2500	0.04

Table 4.2: Link type probabilities and cost metrics used in leaf areas

4.3 Protocol Simulator

In this study, the Routing Protocol Simulator (RPS), a publicly available tool developed by H. Fukushima [Fukushima], was used to simulate logical OSPF clouds and inject OSPF packets into the Router Under Test (RUT).

RPS uses a UDP socket to communicate with the RUT and takes as input three types of files:

1. A network file, which describes the topology of the OSPF cloud to be simulated.
2. A configuration file, which defines the interfaces with the Router Under Test (RUT).
3. A command file, which specifies the behavior of RPS. It contains a list of commands (e.g. interface up/down, router up/down, network up/down, area up/down, ...etc.) to be performed by RPS after reaching the full adjacency state with the RUT.

RPS can simulate multiple OSPF areas within several routing domains. Thus, it will make the RUT behave as if it is connected to a real OSPF network.

4.4 Quantify

QUANTIFY is a commercial performance measurement tool for C and C++ from Rational Software Corporation, used to generate detailed profile information about program execution. It provides a suite of tools to analyze performance bottlenecks and identify portions of the code that dominate execution time. Quantify was selected for two reasons. First, unlike sampling-based profilers, Quantify's reports do not include any overheads. Second, Quantify does not require access to source code when measuring the overhead of system calls and third-party libraries. Performance measurements are recorded in terms of machine instruction cycles and converted to elapsed times according to the clock rate of the machine [Quantify97].

To monitor the execution of the system, first the code of the RUT is linked with QUANTIFY, which will instrument the object code during linkage. Then at runtime, QUANTIFY collects performance data and produces an execution profile showing performance measurements such as the execution cost of functions and their frequency of calls. The execution profile can be used to find principal bottlenecks in the code and identify which portions need tuning.

Chapter 5

OSPF Protocol Performance Analysis

After developing the performance measurement framework, which was presented in Chapter 4, the next step was to analyze and improve the performance of the OSPF routing software. In order to characterize and improve the performance of the software, the following two steps were followed:

Step 1: Identify long and continuous activities that are frequently executed, then analyze them.

There are different ways of measuring performance. One can measure the end-to-end performance, using response time or throughput. Or on the other hand, simply measure the demands for various resources, such as CPU or I/O resource consumption. The overall system performance can be improved either by managing better the contention for system resources or by reducing the resource demands needed to fulfil the arriving service requests. The second approach was selected, by trying to measure and optimize the CPU execution time for expensive components.

It has been observed in different OSPF implementations that when a router is performing routing table calculations (which is a long and continuous activity), all other important processing such as the timely processing of incoming Hello packets halts [Salkewicz95]. If a router does not process a Hello packet received from a neighbor on time, it will declare that neighbor dead. This will in turn lead to the unnecessarily re-establishment of adjacency, and exchanging link-state database information with that neighbor [Sidhu93]. This behavior will impact network stability and availability. One of the requirements for improving the performance was not to

delay the timely processing of Hello packets, and "long continuous" activities are undesirable for this reason.

The execution costs of long and continuous activities that are frequently executed were measured. Then, more detailed measurements to uncover the most expensive operations within these scenarios were performed. The operations (functions) can be expensive per se, and/or very frequent.

Step 2: Optimize most frequent and expensive components within the scenarios identified in Step 1.

After pinpointing functions consuming the highest percentage of resources, the next step is to have a detailed look into these functions and find ways to optimize them [Jain91]. For example, it was discovered that one function was very expensive because it was using an inefficient data structure to insert elements into a sorted list. By using more specialized and efficient data structures this function was optimized (see Section 6.1). Another function was expensive because it was being invoked frequently. The unnecessary and high number of calls to this function were eliminated by the caching of values that were repeatedly computed and reusing them when needed (see Section 6.2), similar to [Gokhale98].

5.1 Workload Selection

The workload is the most crucial part of any performance evaluation project, and it is possible to reach misleading conclusions if the workload is not properly selected [Jain91]. One way to start workload selection is to view the system under study (i.e., the OSPF software) as a service provider. The system provides a list of services, or processes a list of requests, and making a list of those services (or requests) is one of the first steps in a systematic performance evaluation study [Jain91]. By viewing the OSPF system in a router as a service provider, a list of services can be identified. The services can be mapped to a list of scenarios, which exercise these services (or requests).

In this study, the following list of high-level scenarios were studied, which cover the typical activities in an OSPF router [Moy98b]:

- The reception of a Hello packet from an adjacent router.
- The reception of a Database Description packet from an adjacent router.
- The reception of a Link-State Request packet from an adjacent router.
- The reception of a Link-State Update packet from an adjacent router.
- The reception of a Link-State Acknowledgement packet from an adjacent router.
- Link-state database synchronization with an adjacent router.
- Routing table calculations in the event of a topological change in the OSPF area.

All performance measurements in this chapter were performed on a 60 MHz HP-UX workstation with 98 Mbytes of primary memory.

5.2 Performance Measurements of Basic OSPF Protocol Packets

Table 5.1 shows the execution costs of processing Hello, Database Description (DD), Link-State (LS) Request, Link-State Update, and Link-State Acknowledgement packets.

OSPF Packet Type	Details	Execution Cost (in microsecond)
Hello	Used to discover neighbors	99
First DD	Used to initiate database exchange	145
Last DD	Used to terminate database exchange	267
LS Request	Used to request a single LSA	132
LS Update	An update for a single LSA	161
LS Acknowledgement	Acknowledgement for a single LSA	112

Table 5.1: Execution cost of various OSPF packets

Quantifying the execution cost for processing a DD packet is more complex than the other types of OSPF packets. A DD packet is transmitted as part of the link-state database exchange process, and its execution cost varies according to the current state of the exchange process. DD packets can be divided into three types: the first DD packet which is used to initiate the database exchange process, one or more intermediate DD packets which are used to carry LSA summary information (i.e., the LSA header), and the last DD packet which is used to terminate the database exchange process. Table 5.1 shows the execution cost of the first and last DD packets. The execution cost for processing an intermediate DD packet varies with the number of LSA summaries it contains as shown in Figure 5.1.

Figure 5.1: Cost of processing a DD packet as a function of the number of LSA summaries it contains

The execution cost of a Link-State update packet varies with the number of LSAs it contains as shown in Figure 5.2.

Figure 5.2: Cost of processing a Link-State Update packet as a function of the number of LSAs it contains

5.3 Performance Measurements of the Link-State Database Exchange Process

In this section, the performance of the link-state database exchange activity between the RUT and an adjacent router (RPS) is studied. In OSPF networks, every pair of adjacent routers is required to synchronize their link-state databases. To reduce the cost of the link-state database exchange process, especially in broadcast networks, not every two neighboring routers become adjacent. There are certain conditions used to determine which neighbors will be adjacent and therefore will synchronize their link-state databases. A router maintains a finite state machine describing the state of the conversation being held with each one of its neighbors [Moy98b]. Figure 5.3 shows this state machine.

The adjacency starts to form when the neighbor is in state *ExStart* (exchange start). After the two routers agree on their master/slave roles in the exchange process, the state transitions to *Exchange*. One of the two routers becomes the master, and polls the other for database information. The slave returns one or more database description packets that tell about the most recently received information for each LSA in the Link-State database. At this point the two routers start synchronizing their databases. When the synchronization activity is completed, the state transitions to *Full* and we say that the two routers are fully adjacent. Thus, the database exchange process occurs between states *ExStart* and *Full*.

Figure 5.3: The OSPF neighbor finite state machine

Figure 5.4 shows the execution cost of link-state database exchange activity between the RUT and an adjacent router.

Figure 5.4: The performance of the link-state database exchange activity as a function of the number of LSAs exchanged

From Figure 5.4 we see that the link-state database exchange process is an expensive activity. This is due to the inherent complexity of the synchronization process and the sheer number of packets that are exchanged during this process. The two routers first exchange DD packets containing summaries of their link-state databases. Then, each router examines the received summaries, and if there is a set of LSAs in which the router is missing, then the router requests these LSAs by sending one or more Link-State Request packets. The other router responds by sending a Link-State Update packet containing the requested LSAs. Then, the receiving router acknowledges the reception by sending a Link-State Acknowledgement packet. In summary, this complex process involves the transmission of many DD, LS Request, LS Update, and LS Acknowledgement packets.

The link-state exchange process is a very CPU intensive activity. However, the database exchange between two neighboring routers is not a very frequent activity. It only happens when the two routers are establishing adjacency. After the adjacency has been established between the two routers, their link-state databases remain synchronized through the flooding procedure [Moy98b].

5.4 Performance Measurements of the Routing Table Calculation Process

The routing table calculation process is known to be one of the most CPU intensive activities in OSPF [Moy98a, Moy91]. This calculation may cause the longest delays in obtaining new routes after a link/node failure. It is important to update the routing table quickly and provide fast convergence in the event of failures.

It has been observed in different OSPF implementations that when a router is performing the routing table calculations, all other important processing such as the timely processing of incoming Hello packets halts [Salkewicz95]. If a router does not process a Hello packet received from a neighbor on time, it will declare that neighbor dead. This will in turn lead to the unnecessarily re-establishment of adjacency, and exchanging link-state database information with that neighbor [Sidhu93]. This behavior will impact network stability and availability. Improving the performance of the routing table calculation process will help avoid the above mentioned situation, and in addition lead to fast convergence times.

Three high-level scenarios were studied which cover all the different stages involved in the routing table calculation process, as shown in Figure 5.5. These three scenarios are intra-area routes calculations, inter-area routes calculations, and external routes calculations (i.e., routes to destinations outside the Autonomous System) [Moy98b].

In this section we will measure the responsiveness of UMD OSPF software in response to network dynamics. More specifically, we measure how much time it will take OSPF to respond to a change occurring in the same OSPF area (Dijkstra's algorithm), a change occurring in a different area within the same autonomous system, and a change occurring outside the autonomous systems.

Figure 5.5: Conceptual view of the different routing hierarchies supported by OSPF

5.4.1 Intra-Area Routing Table Calculation (Dijkstra's Algorithm)

The complexity of Dijkstra's algorithm is reported in the literature to be of order (n^2) in dense areas, where n is the number of nodes, and $(n \cdot \log n)$ in sparse areas [Moy91, Weiss96]. Algorithm complexity characterizes the behavior of different algorithms at extreme loads, and provides asymptotic results regarding the performance of the algorithm at the extreme loads [Weiss96, Ahuja93]. Therefore, algorithm complexity is an indication of how well the algorithm scales up, but it does not describe in enough detail all the factors contributing to the performance of the implementation at realistic loads (instead of extreme loads) around the operating point of the system. Detailed measurements under different network topologies and workloads are necessary to identify the actual sources of performance bottlenecks and to eliminate them.

In the following set of experiments, the impact of the network topology on the convergence time of Dijkstra's algorithm is studied. Convergence time is defined as the time taken by the RUT to recalculate its routing table as a result of a change in the area to which the RUT belongs. A topological change is introduced by bringing down/up a link. Then from the link-state database, the router rebuilds the routing table by calculating a new shortest path tree, with the root of the tree being the router itself. This algorithm will be referred to as Dijkstra's Shortest Path First (SPF) algorithm [Dijkstra59].

In order to accurately characterize the performance of the algorithm, two different types of experiments were performed. The purpose of the first experiment is to study the effects of network connectivity level on the performance of the algorithm. *Network connectivity* is defined as the average number of links per router in the area. The purpose of the second experiment is to study the effects of number of routers on the performance of the algorithm.

In each experiment, we consider two different types of areas: backbone areas and leaf areas. Backbone areas are characterised by having a high level of connectivity and a low number of routers. Leaf areas are characterised by having a low level of connectivity and a high number of routers. In general, the higher the connectivity level within an area, the denser the area becomes. Therefore, a backbone area generally has a dense topology, and a leaf area has a sparse topology. The density of an area can be defined as the ratio of the number of actual links in the area to the maximum possible number of links. For a fully connected area of N nodes, there will be $N(N-1)$ unidirectional links and the area is called 100% dense. If the ratio is small (i.e., less than 0.20), the area is said to be sparsely connected [Jarvis82].

The purpose of the four experiments are summarized as follows:

1. Study the effect of increasing connectivity level on the performance of the algorithm in a backbone area (with 60 routers).

2. Study the effect of increasing connectivity level on the performance of the algorithm in a leaf area (with 150 routers).
3. Study the effect of increasing the number of routers on the performance of the algorithm in a backbone area (with a connectivity level of 6).
4. Study the effect of increasing the number of routers on the performance of the algorithm in a leaf area (with a connectivity level of 3).

The network parameter ranges used in the experiments were selected based on the characteristics common to today's OSPF networks [Thomas98, Salkewicz95]. The results of the four experiments are shown in Figures 5.6 through 5.9. In all figures, the x-axis represents the number of nodes inside the area, while the y-axis represents the execution cost. The total number of nodes inside an area is defined as the sum of the number of routers and networks inside this area [Moy98b].

From analyzing Figures 5.6 through 5.9 we observed the following:

- The execution cost of Dijkstra's algorithm is directly proportional to the number of nodes inside an area. It increases as the connectivity level increases (as shown in Figure 5.6 and 5.7). It also increases as the number of routers increases (as shown in Figure 5.8 and 5.9).
- For the same number of nodes, the execution cost of the algorithm can change significantly according to the topology of the area. For example, it takes about 25 msec to perform SPF computations on a 400-node area with 60 routers and connectivity level of 26 (Figure 5.6), however it takes less than 15 msec to perform SPF computations on a 400-node area with 150 routers and a connectivity level of about 6 (Figure 5.7). This illustrates that increasing the connectivity level within an area has a greater effect on the execution cost of Dijkstra's algorithm than increasing the number of routers inside the same area.
- The cost curve has a linear shape when the connectivity level is low (i.e., the area is sparse). This behavior can be seen in Figure 5.9. As the connectivity level increases, the area becomes denser and the shape of the graph changes to quadratic. This behavior can be observed in Figure 5.6.

Figure 5.6: The effect of network connectivity on Dijkstra's algorithm in a backbone area

Figure 5.8: The effect of number of routers on Dijkstra's algorithm in a backbone area

Figure 5.9: The effect of increasing the number of routers on Dijkstra's algorithm in a leaf area

5.4.2 Inter-Area Routing Table Calculation

In this experiment we measured the execution cost of inter-area routing table calculations due to topological changes in a different area within the autonomous system. We increased the number of summary LSAs (inter-area destinations) from 500 to 1600 and measured the cost of inter-area routing table calculation. Figure 5.10 shows the results for this experiment. It should be mentioned that Dijkstra's algorithm is not used in the inter-area routing table calculations.

The execution cost graph has a linear shape because exchanging routing information between areas is essentially a Distance Vector algorithm. For any given inter-area destination, the calculating router examines all summary LSAs advertising that destination, and uses the best summary LSA to create a routing table entry for that destination. Thus, topological changes in other areas do not require complete recalculation of the entire routing table. The reader is referred to [Moy98a, Moy98b] for a detailed discussion on this subject.

Figure 5.10: The cost of inter-area routing table calculation

5.4.3 External Routing Table Calculation

Finally in this experiment we measured the execution cost of external routing table calculations due to topological changes outside the autonomous system (AS). We increased the number of external LSAs (inter-AS destinations) from 400 to 1000 and measured the cost of external routing table calculation. Figure 5.11 shows the results for this experiment. Similar to the last experiment, we find that the execution cost has a linear shape in this experiment.

Similar to the experiment in Section 5.4.2, the execution cost graph has a linear shape because exchanging routing information between autonomous systems is also a Distance Vector algorithm. For any given destination outside the autonomous system, the calculating router examines all external LSAs advertising that destination, and uses the best external LSA to create a routing table entry for that destination.

Figure 5.11: The cost of external routing table calculation

Summary of Observations:

- The execution cost of Dijkstra's algorithm is a function of the number of nodes (routers and networks) in the OSPF area. The algorithm is more sensitive to the connectivity level inside an OSPF area than to the number of routers.
- The shortest path calculation due to a topological change within an area is the most expensive component in the routing table calculations. This cost is much higher than the cost of calculations induced by changes in other areas or other autonomous systems. There are two reasons for this. The first reason is that summary and external LSAs are much smaller than router and network LSAs. The second reason is that topological changes in other areas and autonomous systems do not require recalculation of the entire routing table. However, it is always necessary to recalculate the entire routing table for any topological changes within the OSPF area [Moy98b].

5.4.4 Detailed Analysis: Identification of the Most Expensive Functions

To understand where the time is being spent in the implementation of Dijkstra's algorithm, two detailed experiments were performed to analyze the execution behavior of the functions called by the algorithm's implementation.

The purpose of the first experiment is to study the performance of the functions in a dense topology, or in other words, to study the effect of increasing the connectivity level inside an area on the performance of the functions. The purpose of the second experiment is to study the performance of the functions in a sparse topology, or in other words, to study the effect of increasing the number of routers inside an area on the performance.

Figure 5.12 shows the performance of the top most expensive functions, in a dense topology, as the connectivity level increases. Figure 5.13 shows the performance of the top most expensive functions, in a sparse topology, as the number of routers increases. In both figures, the x-axis represents the number of nodes inside an area, which is the sum of number of routers and networks inside the area. The y-axis represents the total execution cost, which is the direct product of two parameters: the frequency of calls to a function and the time consumed in each call.

Figure 5.12: The effect of increasing network connectivity on the performance of Dijkstra's algorithm (detailed view)

When the connectivity level increases in an area (i.e. when the topology is getting denser), as shown in Figure 5.12, we observed the following:

- The growth of all top functions is linear except for the **cl_enq** (candidate list enqueue) function, which has a quadratic growth.

- The **findLSA** function is the most expensive function as long as the connectivity level is less than 8. As the connectivity level increases beyond 8, the **cl_enq** function becomes the most expensive (bottleneck) function.

Figure 5.13: The effect of increasing number of routers on the performance of Dijkstra's algorithm (detailed view)

The next step is to have a detailed look into the most expensive functions and find ways to optimize them.

The implementation of Dijkstra's algorithm uses **cl_enq** function to insert a new vertex into the candidate list. The candidate list is implemented as a sorted linked list of LSDB data structures (as shown in Figure 5.14). The **cl_enq** function linearly searches the list every time a new vertex needs to be inserted.

Figure 5.14: A conceptual view showing the insertion of a new vertex into the candidate list.

To better characterize the performance and evaluate the efficiency of the **cl_enq** function, more performance data (metrics) were collected during its execution. Table 5.2 shows the performance metrics, for the **cl_enq** function, collected during one execution of Dijkstra's algorithm in a backbone area with 60 routers and connectivity level of 26. We see that **cl_enq** contributes 33% to the total execution cost of the implementation of Dijkstra's algorithm.

Function Name	Total Time (clock cycles)	# calls	Ratio of Dijkstra	Min Time	Max Time
cl_enq	469857	391	33%	40	4859

Table 5.2: Performance data for the **cl_enq function**

As shown in Table 5.2, the execution cost of inserting a new vertex into the candidate list is highly variable. It ranges from 40 clock cycles to 4859 clock cycles, and is dependent on the length of the candidate list and the value of the cost metric associated with the vertex being inserted. The execution cost of the function is minimal when inserting at the beginning of the list and maximum when inserting at

the end of the list. The insertion algorithm is a good candidate for optimization because it consumes a significant fraction of total execution time.

When the number of routers increases in a sparse area, as shown in Figure 5.13, we observed that the execution cost of most functions is linear, and that the **findLSA** function is always the most expensive function.

The implementation of Dijkstra's algorithm uses the **findLSA** function to find a vertex (LSDB data structure) with a particular link-state id (LS_ID). The function first performs a hashing calculation on the given LS_ID to get a hash key index into the LSDB hash table. Then, if collisions occur (i.e. two different link-state ids have the same hash key), the function will search the linked list linearly at the hash key position to find the LSDB structure with the correct link-state id. Figure 5.15 shows a conceptual view of this process. High collision rates can be due to an inefficient hashing function and/or a small hash table size. Resolving such collisions takes time, hence reduces the efficiency of the hashing function.

Figure 5.15: A conceptual view of how the findLSA function works

Table 5.3 shows the performance data, for the **findLSA** function, collected during one execution of Dijkstra's algorithm in a leaf area with 350 routers and connectivity level of 3. We see that the **findLSA** function contributes about 15% to the total execution cost of Dijkstra's algorithm.

Function Name	Total Time (clock cycles)	# calls	Ratio of Dijkstra	Min Time	Max Time
FindLSA	221182	1681	15%	89	191

Table 5.3: Performance data for the findLSA function

Based on above the observations, the following performance limiting factors within the implementation of Dijkstra's algorithm are identified:

1. The use of inefficient data structures for the candidate list. Currently, the candidate list is implemented using a linked list. The **cl_enq** function linearly searches the linked list when inserting a new vertex. The insertion operation has a time complexity of $O(n)$, where n is the number of nodes in the list.
2. The high frequency of calls to the **findLSA** function, which is used to retrieve a vertex (LSDB structure) from the link-state database.

Chapter 6

Solutions for Performance Improvement

In this chapter, two proposed solutions to optimize the frequent and expensive functions identified in the last chapter are presented.

6.1 Optimization Solutions for the `cl_enq` Function

From the detailed analysis of the implementation of Dijkstra's algorithm in Section 5.4.4, it was found that the `cl_enq` function is one of the most expensive components. This is because a linked list was used to implement the candidate list, and the `cl_enq` function linearly searches the list in order to insert a new vertex in a sorted order.

There many different types of data structures reported in the literature which can be used to implement the candidate list, like binomial heaps, Fibonacci heaps, splay trees, and binary heaps [Weiss96, Ahuja93]. Oberhauser et al. performed a comparative evaluation of these data structures, and found that a binary heap outperformed the more complex data structures when used to implement the candidate list [Oberhauser95]. Figure 6.1 shows the performance of Dijkstra's algorithm using three different data structures to implement the candidate list. We see that a simple binary search tree (which was not part of Oberhauser's study) used to implement the candidate list is more efficient than either the linked list and the binary heap.

Figure 6.1: Performance measurements of Dijkstra's algorithm using different data structures to implement the candidate list

6.2 Optimization Solutions for the findLSA Function

From the detailed analysis of the implementation of Dijkstra's algorithm in Section 5.4.4, it was also found that the **findLSA** function is one of the most expensive components. This is due to the repeated and high number of calls to this function. A substantial reduction in the execution cost of the implementation of Dijkstra's algorithm can be achieved by finding a way to reduce the high number of calls to the **findLSA** function.

As was previously mentioned in Section 2.2, when a new node is to be added to the shortest path tree, Dijkstra's algorithm examines all of its neighbors for possible inclusion in (or modification of) the candidate list. The **findLSA** function is used to retrieve the next neighbor from the link-state database. If Dijkstra's algorithm runs again, and assuming there are no changes to the link-state database, then **findLSA** will be called in the same fashion again. Since the neighbors of each node have not changed, the calls to **findLSA** in the subsequent runs of Dijkstra's algorithm can be

eliminated by caching the pointers to these neighboring nodes, which were obtained from calls to **findLSA** during the previous execution of Dijkstra's algorithm.

The high number of unnecessary calls to **findLSA** were eliminated by using more efficient data structures that facilitate caching of pointers to neighboring nodes. Accordingly, pointers to neighboring nodes will be calculated once and cached for subsequent accesses. Figure 6.2 compares the performance of the original Dijkstra's algorithm implementation and the caching solution. Significant gains in performance were obtained by caching of pointers and reducing the repeated calls to the **findLSA** function. Appendix B describes the design and the implementation issues of this solution.

Figure 6.2: Performance comparison of the original implementation of Dijkstra's algorithm and an optimized implementation using the caching method

The two proposed solutions are independent of each other, and can be combined to produce a more efficient implementation of Dijkstra's algorithm. Figure 6.3 compares the performance of Dijkstra's algorithm incorporating the two proposed solutions, and the original implementation of the algorithm. The proposed solutions have resulted in significant gains in performance. For example, by using these techniques, a 40% improvement in the performance of routing table calculation activity was achieved (in a backbone area with 400 nodes and a connectivity level of 26).

Figure 6.3: Performance comparison of the original implementation of Dijkstra's algorithm and an optimized implementation incorporating the two proposed solutions

Chapter 7

Design and Implementation of the ISPF Algorithm

In the last chapter, the performance of Dijkstra's algorithm implementation in the UMD OSPF software was improved by optimizing some of the key data structures used extensively during the routing table calculation activity. This chapter shows that significant gains in performance can be achieved by replacing Dijkstra's algorithm by a more efficient algorithm.

It was mentioned in Section 2.3 that Dijkstra's Shortest Path First (SPF) algorithm has a critical deficiency, since it requires a complete recalculation of the routing table as a result of changes to the OSPF area's topology. The Incremental SPF algorithm (ISPF) should be more efficient than the standard Dijkstra's SPF algorithm because it requires only incremental calculations to the routing table in the event of topological changes

This chapter describes the design of the ISPF algorithm that was incorporated into the UMD OSPF software. Section 7.1 discusses the data structures used in the algorithm's implementation. Section 7.2 outlines the different components (modules) in the implementation and how they interact. Finally, Section 7.3 presents performance analysis of the ISPF algorithm. Appendix C provides some illustrative scenarios on the effects of topological changes within an OSPF area, and how the ISPF algorithm handles them.

7.1 Data Structures used in the Design

7.1.1 The Link-State Database

The link-state database is the collection of all OSPF Link-State Advertisements (LSAs) in an area, which gives a complete description of the topology (the routers and network segments in the area and how they are interconnected). A router has a separate link-state database for every area to which it belongs. Link-state databases are exchanged between neighboring routers soon after the routers have discovered each other; after that the databases remain synchronized through a procedure called reliable flooding.

7.1.2 The SPF Tree Data Structure

Dijkstra's SPF algorithm generates a tree representing the minimum delay paths from a given source node to every other node in the area. This tree is called the SPF tree. Figure 7.1 shows an example of a SPF tree, and Figure 7.2 depicts a pointer-based implementation of the SPF tree. The implementation uses a general tree data structure commonly found in the literature [Carrano95]. Each node in the tree has two essential pointers: a pointer to the node's oldest child, and a pointer to the node's next (younger) sibling. There are two more pointers that are not necessary, but are used to speedup the tree traversal and manipulation operations: a pointer to the node's parent and a pointer to the node's oldest sibling.

The tree data structure supports a set of common operations needed in ISPF: remove or relocate a subtree rooted at a node, increase/decrease the cost of all node in a subtree by a constant value, and so on.

Figure 7.1: A sample SPF tree **Figure 7.2: A pointer-based implementation**

The most of the memory requirements of the ISPF algorithm comes from the need to maintain the SPF tree. Dijkstra's SPF algorithm, as it is implemented in the UMD OSPF software, doesn't save the SPF tree, but rather it directly updates the routing table whenever a new path is found to each destination

7.1.3 The Topology Change List

The topology change list is a linked list of change events. Each event corresponds to the occurrence of a particular topological change inside the OSPF area. The basic idea is that when a router receives a LSA, it tries to find out the kind of event that triggered the generation of this LSA. This is achieved by comparing the new LSA, which was generated by some node in the OSPF area, with the node's last installed LSA in the link-state database, if it exists. Once the receiving router determines the exact type of change, it generates a new change event and adds it to the topology change list. All the topological changes are chained in a single linked list. Table 7.1 shows the types of topological changes in an OSPF area.

Event Name	Description
LinkDown	Failure of a communication link
LinkUp	Recovery of a communication link
LinkCostIncrease	Cost increase of a communication link
LinkCostDecrease	Cost decrease of a communication link
NewLSA	The arrival of a new LSA which has not been previously received
AgedLSA	Some LSA reaching its maximum allowable age (expiring)

Table 7.1: Types of topological changes in an OSPF area

The *LinkDown* event represents the failure of a communication link between two routers (or between a router and a network) inside the OSPF area. The *LinkUp* event represents a communication link between two routers (or between a router and a network) inside the OSPF area becoming functional. The *LinkCostIncrease* and *LinkCostDecrease* events represent whether the cost of the link has increased or decreased. The *NewLSA* event represents the arrival of a new LSA, which was originated from a newly operational node (router or a network), and has not been previously received.

The last event in the table, *AgedLSA*, is a more special event. The maximum allowable age for an LSA is one hour [Moy98b]. The age of an LSA is periodically incremented, and if an LSA exceeds its maximum allowable age, it will be deleted from the link-state database (the topology graph). The *AgedLSA* event is different from the other events in the following way. A router generates the other types of change events as a result of *receiving* an LSA from some router in the area. However, the *AgedLSA* event is generated when a locally stored LSA exceeds its maximum allowable age, and not as a results of receiving an externally generated LSA.

Some events are more common than others. For example, the occurrence of *LinkDown*, *LinkUp* and *NewLSA* events is more likely than the occurrence of *LinkCostIncrease* and *LinkCostDecrease* events. Dynamically changing link costs is

not common [Moy98a], but nevertheless, it is possible, if a network administrator decides to dynamically change the link costs of a router's active interfaces.

7.1.4 The Candidate List

The candidate list structure is used by Dijkstra's algorithm to contain the nodes that are potential candidates to be added to the shortest-path tree in the next iteration of the algorithm (see section 2.2). In UMD OSPF, the candidate list is implemented as a linked list of nodes that are sorted based on their distances from the source router (i.e., the calculating router).

7.1.5 The Routing Table

The routing table is derived from the link-state database, and contains all the information necessary to forward an IP data packet toward its destination. Each entry in the routing table describes the collection of best paths to a particular destination. When forwarding an IP data packet, the routing table entry providing the best match for the packet's IP destination is located. The matching routing table entry then provides the next hop towards the packet's destination.

7.2 The ISPF Algorithm Components

The ISPF algorithm implementation is composed of the different components shown in Figure 7.4. In this figure, information flow through the system is illustrated using the Data Flow Diagram (DFD) graphical notation [Pressman92].

The following is a description of each component:

The Topology Change Detector

A router receives five different types of OSPF packets (Hello, Database Description, Link-State Request, Link-State Update, and Link-State Acknowledgement) from neighboring routers. If the received packet is a Link-State Update packet, then the Topology Change Detector component will be called in order to analyze the incoming LSAs contained within this packet.

The Topology Change Detector implements the logic that identifies the changes that have occurred in the area's topology. It analyzes every incoming LSA, which was generated by some node in the OSPF area, and compares it with the node's last installed LSA in the link-state database, if it exists. For example, Figure 7.3 shows two LSAs originated from router R4. LSA 1 is already installed in the link-state database of router R1, and LSA 2 is a newer instance that was just received by R1. By comparing the two LSA instances, the Topology Change Detector will determine that the link from R4 to R6 is down. The other types of topological changes can be identified in a similar manner.

Figure 7.3: Two different instances of the LSA generated by router R4

Once the Topology Change Detector determines the exact type of topological change, it generates a new change event and adds it to the topology change list. All the topological changes are chained in a single linked list.

The OSPF Interface Manager

The OSPF Interface Manager is responsible for maintaining the state of interfaces (links) and neighbors of the router and detecting their failure or recovery. There are two common mechanisms used to detect the failure or recovery of an interface or a neighbor. The first mechanism is interface status changes (such as carrier signal loss or excessive framing errors on a serial link) which is used to indicate that an interface is no longer functional. The second mechanism is the failure of the router to receive a Hello packet from its neighbor within a timing window called the dead timer interval (usually 40 seconds). After this timer expires, the router assumes the neighbor is down [Thomas98].

In the event of an interface/neighbor status change, the OSPF Interface Manager creates a new LSA describing the topological change, installs it in the local link-state database, and floods the LSA throughout the area. Then, the OSPF Interface Manager creates a new change event describing the topological change and adds it to the topology change list.

Figure 8.4: The ISPF Algorithm Components

The ISPF Algorithm

When the ISPF algorithm component gets invoked, it iterates through the list of change events in the topology change list and examines each change event. The algorithm tries to identify and update only the shortest-path routes affected by the topological change. Figure 7.5 shows a high level flowchart for the ISPF algorithm incorporated into the UMD OSPF software.

Figure 7.5: High-level flowchart for the ISPF Algorithm

Assume that S is the node (router) calculating the routing table, and it is the root of the SPF tree. The following is a list of the types of topological change events in an OSPF area, and how ISPF processes them:

- **LinkDown (from node A to node B) Event:**

If this link is not in the SPF tree, then do nothing. Else, if the link is in the SPF tree, then isolate all the nodes in the subtree rooted at B and set their path costs from S to infinity. Then, for each node N of the subtree, try to find a shorter path from S to N via an adjacent node that is not already in the subtree. If such a path

is found, then add N to the Candidate List. If still there are nodes with path costs of infinity, this means that these nodes are unreachable, and the routing paths to these nodes should be removed from the routing table.

- **LinkUp (from node A to node B) Event:**

If the cost of using the new link results in a shorter path to node B (i.e., the cost of the new path is lower than the cost of the old path to B), then the best route to reach B will pass through A. The first change to the SPF tree is, therefore, to relocate B (and its subtree), attaching it to node A via the improved link. Then, isolate all the nodes in the subtree rooted at B and update their path costs from S. Next, for each node N that is not in the subtree but is adjacent to a subtree node, try to find a shorter path to N via an adjacent node which is in the subtree. If such a path is found, then add N to the Candidate List.

- **LinkCostIncrease (from node A to node B) Event:**

If this link (whose cost was increased by X units) is not in the SPF tree, then do nothing. Else, if the link is in the SPF tree, isolate all the nodes in the subtree rooted at B and increase their path costs from S by X units. Then, for each node N of the subtree, try to find a shorter path to N via an adjacent node that is not already in the subtree. If such a path is found, then add N to the Candidate List.

- **LinkCostDecrease (from node A to node B) Event:**

If this link (whose cost was increased by X units) is in the SPF tree, isolate all the nodes in the subtree rooted at B and decrease their path costs from S by X units. Then, for each node N that is not in the subtree but is adjacent to a subtree node, try to find a shorter path to N via an adjacent node which is in the subtree. If such a path is found, then add N to the Candidate List.

If the link is not in the SPF tree, then if the cost of the new path from S to B using the improved link is greater or equal than the cost of the original shortest path to B, then B can not take advantage of this improvement, and there will be no

changes to the SPF tree or to the routing table. If, on the other hand, the new path is shorter than the original path, then the best route to reach B will pass through A. The first change to the SPF tree is, therefore, to relocate B (and its subtree), attaching it to node A via the improved link. Now the situation is identical to that of the previous paragraph in which the link from A to B was in the tree in the first place and its cost decreased.

- **NewLSA Event:**

Try to find the shortest path from S to node B, which originated the new LSA, through one of B's neighbors. If such a path is found, then add B to the Candidate List.

- **AgedLSA Event:**

Remove the node M whose LSA has aged out (expired) from the link-state database and from the routing table. If there is a subtree rooted at M, then isolate all the nodes in the subtree and set their path costs from S to infinity. Then for each node N of the subtree (except M of course), try to find a shorter path to N via an adjacent node which is not already in the subtree. If such a path is found, then add N to the Candidate List. If still there are nodes with path costs of infinity, these nodes are unreachable and the routing paths to these nodes should be removed from the routing table.

The SPF Algorithm

The SPF algorithm component, which implements a slightly modified version of Dijkstra's SPF algorithm, is used to update the routing table after the topological change has occurred in the OSPF area.

The standard SPF algorithm builds the routing table by first adding the source (or the calculating) router to the SPF tree. Then it adds all of the source router's neighbors to the Candidate List. Instead of starting from an empty Candidate List, as does the

standard SPF algorithm and proceeding from there, the slightly modified version of the SPF algorithm starts from the Candidate List populated by the ISPF algorithm component. Then it proceeds in the regular fashion by removing the node on the Candidate List with the smallest distance from the source router and adding it to the SPF tree. Then that node's neighbors are examined for possible inclusion in (or modification of) the Candidate List. The algorithm terminates when the Candidate List becomes empty. Figure 7.6 shows a high-level flowchart for the SPF algorithm incorporated into the UMD OSPF software.

Figure 7.6: High-level flowchart for the modified SPF Algorithm

7.3 Performance Measurements

The goals of the two performance experiments presented in this section are to compare the execution costs of the ISPF and the traditional SPF algorithms, and quantify the performance improvements that resulted from using ISPF. The comparison was performed using both backbone and leaf area topologies.

In each of the two experiments, the time taken by the RUT to recalculate its routing table as a result of a change in the OSPF area was measured. A topological change is introduced by failing a router inside the area. Then if the RUT is using Dijkstra's SPF algorithm, the entire routing table for this area is recalculated. If, on the other hand, the RUT is using the ISPF algorithm, then only those routing table entries affected by the topological change will be updated.

All performance measurements were performed on a 60 MHz HP-UX workstation with 98 Mbytes of primary memory.

The results of the experiments are shown in Figures 7.7 and 7.8. In both figures, the x-axis represents the number of nodes inside the area, while the y-axis represents the execution cost. The total number of nodes inside an area is defined as the sum of the number of routers and networks inside this area [Moy98b].

Figure 7.7: The total execution cost of the ISPF and SPF algorithms in a backbone area

Figure 7.8: The total execution cost of the ISPF and SPF algorithms in a leaf area

From analyzing Figures 7.7 and 7.8, it was observed that the execution cost of the ISPF algorithm is much less than the execution cost of Dijkstra's SPF algorithm. The reason is that for every change in the area, SPF requires a complete recalculation of the entire routing table. However, ISPF eliminates the full routing table calculations in favor of only partially rebuilding the necessary portions of SPF tree.

In addition to the previous observation, it was found that the execution cost of Dijkstra's SPF algorithm is directly proportional to the *total* number of nodes inside the OSPF area. However, the execution cost of ISPF is directly proportional to the number of nodes affected by the topological change (the size of the changed subtree), which leads to a better performance. This result is in agreement with the findings in [McQuillan80] and can be explained as follows. From the background discussion about ISPF in Section 2.3, we know that whenever a topological change occurs, the algorithm will try to isolate (or identify) the group of nodes which are affected by the change. In some cases, the topological change will affect a small number of nodes, and hence the execution cost of ISPF to update the shortest paths to these nodes will be minimal. In some other cases, the topological change will affect a large number of nodes, and will lead to a large number of routes to be recalculated. This will result in a larger execution cost than in the previous case.

We said that the execution cost of ISPF is proportional to the size of the affected subtree, and as the size of the subtree gets larger and larger, the execution cost of ISPF approaches the execution cost of SPF. Because of this fact, it is possible, at least in theory, to have a topological change that affects all the nodes in the OSPF area, and leads to an ISPF execution cost which is as bad as the execution cost of SPF. In practice, this scenario is very rare to happen in real networks. From the experiments that were conducted, it was found that most subtrees have a small number of nodes. In other words, most of the time a topological change will affect a small number of nodes, this means that a few number of routes need to be updated in the routing table, and a short execution time for ISPF.

Figure 7.9 depicts a histogram showing the distribution of the subtree sizes and their frequency of occurrence in a Backbone area with a connectivity level of 26. In this OSPF area, there were 60 routers and 344 edges, leading to a total of 404 network elements. We see that the majority of the subtree sizes are under 200 nodes, and very few are more than 200 nodes. In order to see the details of the histogram better, a smaller bin size is used, as shown in Figure 7.10. We see that out of a total of 60 subtrees, 41 subtrees contained less than 15 nodes.

Figure 7.9: Large-scale histogram of ISPF subtree sizes

Figure 7.10: Small-scale histogram of ISPF subtree sizes

Figure 7.11 presents a detailed view of the results shown in Figure 7.7 comparing the execution costs of SPF and ISPF in a backbone area with 60 routers while varying the connectivity level. We see that the average ISPF execution cost in the event of a topological change is close to the minimum observed execution cost of ISPF in this experiment. The reason is that the average subtree size was very close to the minimum observed subtree size. In addition, we see that the maximum observed ISPF execution cost was less expensive than the execution cost of SPF.

Figure 7.11: Detailed comparison between the execution costs of ISPF and SPF algorithms in a backbone area

In conclusion, the SPF algorithm takes a fixed amount of time to perform the routing table calculation for a given size and connectivity. For example, it was found that the SPF algorithm takes about 20 ms for a backbone area of about 400 nodes and a connectivity level of 26. This is an expected result since the SPF algorithm does a complete routing table calculation in the event of a topological change. On the other hand, the execution cost of the ISPF algorithm varies and is proportional to the number of updated shortest-path routes necessitated by the topological change.

Chapter 8

Conclusions

The general objectives of this research were to understand the performance characteristics of the UMD OSPF software, to identify bottlenecks in the implementation, and to propose efficient solutions to alleviate their effects and improve the performance of the system.

In order to meet these objectives, here is a list of the general steps followed:

- 1- Understand the present system by reverse engineering the implementation.
- 2- Study and characterize the performance of the current system, and discover bottlenecks in the implementation.
- 3- Propose and implement solutions to remedy the effects of the bottlenecks by using efficient data structures and algorithms.
- 4- Measure the performance of the proposed implementation and compare it with the original implementation.

The proposed solutions have resulted in significant performance gains. For example, by using the two solutions presented in Chapter 6, 40% reduction in the execution time of the routing table calculation activity (in a backbone area with 400 nodes) was achieved. An 87% reduction in the execution time of the routing table calculation activity (in a backbone area with 400 nodes) achieved by replacing the standard Dijkstra's algorithm with the ISPF algorithm was also observed.

Various Nortel Networks routing products are currently using the proposed solutions. In one case, it was observed that the overall utilization of the router-under-test dropped by 50% when running the improved OSPF software, as compared to running

the original software (under the same workload). This illustrates that a 50% overall reduction in CPU execution time was achieved.

Summary of Conclusions

During the course of this research, we reached the following conclusions:

- The results show that Dijkstra's algorithm, which is used in the routing table calculation process, is one of the major sources of overhead in OSPF.
- Algorithm complexity characterizes the behavior of different algorithms at extreme loads, and provides asymptotic results regarding the performance of the algorithm at the extreme loads [Weiss96, Ahuja93]. Therefore, algorithm complexity is an indication of how well the algorithm scales up, but it does not describe in enough detail all the factors contributing to the performance of the implementation at realistic loads (instead of extreme loads) around the operating point of the system. Detailed measurements under different network topologies and workloads are necessary to identify the actual sources of performance bottlenecks and to eliminate them.
- The choice of the underlying data structure used to implement the Candidate List is crucial for performance of the implementation of Dijkstra's algorithm. Significant computational improvements have resulted from using data structures that implement the Candidate List manipulation operations efficiently.
- The performance issues identified in this research are not unique to UMD OSPF. Three other OSPF implementations were examined, and it was found that they have similar deficiencies. The proposed solutions in this thesis, including the ISPF algorithm, are generic and can be used to improve the performance and scalability of other OSPF implementations.

- By using a high-level optimization at the algorithm level, much more gains in performance can be achieved than what is possible from code-level optimization. More gains in performance were obtained from using the ISPF algorithm in the routing table calculation activity, more than what was achieved from optimizing the data structures used by the conventional Dijkstra's algorithm. The performance experiments conducted show that ISPF clearly outperforms the conventional Dijkstra's algorithm in terms of the computational requirements.
- It is important to appreciate that no single reverse engineering tool will suffice for all architectural extraction and analysis of a system. Most tools are designed to extract particular source information rather than comprehensive models, and they frequently produce output that does not accurately reflect the architecture of the system. Many researchers in the field have also reached similar conclusions [Murphy98, Lung98a, Kazman98b]. Chapter 3 presented the various tools that were utilized to recover the architecture of the UMD OSPF software.

Future Work

There many possible venues for a future research, varying in difficulty, effort, and scope. Some deal with how to improve the quality attributes of the software, and other deal with improving its performance. The future work can be divided into the following two general categories:

Research directions to improve the architectural qualities of the software:

There is growing recognition of the importance of the software architecture in determining the quality attributes, such as modifiability, reusability, and reliability of a software system. While a good architecture cannot guarantee attainment of quality goals, a poor architecture can prevent their achievement.

During the course of this research, a systematic process for architecture recovery was followed. While architecture recovery is an important and valuable contribution, we also want to go one step further and improve the architecture rather than simply recover it. The recovered architecture of the OSPF system can form the basis for future studies.

The following is a list of such studies:

1. Study the applicability of Design Patterns to improve the architectural qualities of the system

By examining the implementation and the recovered architecture, the areas that are highly coupled can be identified, and ways to restructure or reengineer the code to improve the architecture can be explored. This can be done through the identification and adoption of appropriate design patterns, similar to the patterns found in [Gamma95], to improve the maintainability of the system, and reduce the coupling between components.

2. Study the flexibility of the software to remove/modify/add features

The UMD OSPF software was developed in 1989. New features are always being added to the original software to make it compliant with new OSPF standards. The software architecture should be evaluated with respect to the property of modifiability to find how flexible is the architecture in accommodating changes.

Research directions to improve the performance and scalability of the software:

1. Capacity planning of routers running the UMD OSPF routing software.

The resource demands for the various services provided by the protocol can be obtained using the performance measurement framework. Then, a layered

queuing network model of an OSPF router can be constructed. This model can be used for capacity planning and scalability analysis of the router. For example, the model can be used to find out the maximum number of virtual routers that can be configured in a single physical router.

2. Optimizing the database exchange process.

The database exchange process between adjacent routers is an expensive activity. The cost of this activity increases as the number of neighbors of a router increases. A detailed study should be done to identify ways to optimize this process.

3. Investigating the performance gains from using Dial's, Radix and other specialized data structures to implement the candidate list.

In this research, Binary Heap and Binary Search Tree data structures were used to implement the Candidate List. There are other data structures reported in the literature worth investigating, like Dial's Data structure [Dial69] and Radix Heap [Ahuja93].

References

- [Ahuja93] Ahuja, R., et al., "*Network Flows: Theory, Algorithms, and Applications*", Prentice Hall, 1993.
- [Anquetil98] Anquetil, N., Lethbridge, T., "*Extracting Concepts from File Names; a New File Clustering Criterion*", International Conference on Software Engineering '98, Japan, pp. 84-93, 1998.
- [Bertsekas87] Bertsekas, D., Gallager, R., "*Data Networks*", Prentice-Hall, 1987.
- [Bruno96] Bruno, C., "*Internet Health Report: Condition Serious*", Network World, pp. 12-14, September 1996.
- [Calvert97] Calvert, K.L., et al., "*Modeling Internet Topology*", IEEE Comm. Magazine, June 1997.
- [Carrano95] Carrano, F., "*Data Abstraction and Problem Solving with C++*", Benjamin/Cummings Publishing Company, Inc., 1995.
- [Comer98] Comer, D.E., Stevens, D. L., "*Internetworking With Tcp/Ip: Design, Implementation, and Internals*", Third Edition, Vol. 2, Prentice Hall, June 1998.

- [Dial69] Dial, R.B., et al., "*Algorithm 360: Shortest-Path Forest with Topological Ordering*", Communications of the ACM, Vol. 12, No. 11, pp. 632-633, November 1969.
- [Dijkstra59] Dijkstra, E.W., "*A Note on Two Problems in Connection with Graphs*", Num. Math., Vol. 1, pp. 269-271, 1959.
- [Doar96] Doar, M., "*A Better Model for generating Test Networks*", Proceedings of Globecom 96, also at <ftp://ftp.nexen.com/pub/papers>, Global Internet 96, November 1996.
- [Fukushima] Fukushima, H., Routing Protocol Simulator
URL:http://www.gated.org/new_web/code/doc/code_info/tools/rps.html
- [Gamma94] Gamma, E., Helm, R., Johnson, R., Vlissides, J., "*Design Patterns: Elements of Reusable Object-Oriented Software*", Reading, Mass., Addison-Wesley, 1994.
- [Gokhale98] Gokhale, A., Schmidt, D. C., "*Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance*", Proceedings of the HICSS conference, Maui, Hawaii, January 9th, 1998.
- [Hedrick88] Hedrick, C., "*RFC 1058: Routing Information Protocol*", June 1988.

- [Huitema95] Huitema, C., *"Routing in the Internet"*, Prentice Hall, 1995.
- [Jain91] Jain, R., *"The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling"*, John Wiley and Sons, 1991
- [Jarvis82] Jarvis, J., *"A Decomposition Algorithm for Locating A Shortest Path Between Two Nodes in a Network"*, Networks, Vol. 12, pp. 161-172, 1982.
- [Kazman98a] Kazman, R., Carriere, S. J., *"View Extraction and View Fusion in Architectural Understanding"*, Proceedings of ICSR5, Victoria, B.C., pp. 290-299, June 1998.
- [Kazman98b] Kazman, R., Carriere, S. J., *"Playing Detective: Reconstructing Software Architecture from Available Evidence"*, Journal of Automated Software Engineering, 1998.
- [Lottor99] Lottor, M., *"Internet Domain Survey"*, Network Wizards, January 1999. URL: <http://www.nw.com/>
- [Lung98a] Lung, C.H., *"Software Architecture Recovery and Restructuring through Clustering Techniques"*, ISAW3, Orlando, Florida, pp. 101-104, 1998.
- [McQuillan78] McQuillan, J., Falk, G., Richer, I., *"A Review of the Development and Performance of the ARPANET Routing Algorithm"*, IEEE Transactions on

- Communications, Vol. 26, No. 12, pp. 802-1811, December 1979.
- [McQuillan80] McQuillan, J., Richer, I., Rosen, E., "*The New Routing Algorithm for the ARPANET*", IEEE Transactions on Communications, Vol. 28, No. 5, pp. 711-719, May 1980.
- [Moy98a] Moy, J., "*OSPF: Anatomy of an Internet Routing Protocol*", Addison-Wesley, February 1998.
- [Moy98b] Moy, J., "*RFC 2328: OSPF Version 2*", April 1998.
- [Moy91] Moy, J., "*RFC 1245: OSPF Protocol Analysis*", July 1991.
- [Murphy98] Murphy, G., Personal communication with G. Murphy, October 1998.
- [Murphy97] Murphy, G., Notkin, D., "*Reengineering with Reflexion Models: A Case Study*", IEEE Computer Magazine, pp. 29-36, August 1997.
- [Petrazzini99] Petrazzini, B., Kibati, M., "*The Internet in Developing Countries*", Communications of the ACM, Vol. 44, No. 6, pp. 31- 36, June 1999.
- [Pressman92] Pressman, R. S., "*Software Engineering: A Practitioner's Approach*", Third Edition, McGraw-Hill, Inc., 1992.

- [Oberhauser95] Oberhauser, G., Simha, R., "*Fast Data Structures for Shortest Path Routing: A Comparative Evaluation*", IEEE International Conference on Communications '95, Vol. 3, pp.1597 -1601, 1995.
- [OSPFGroup] The OSPF Discussion Group, URL: <http://www.ietf.org/html.charters/ospf-charter.html>
- [Quantify97] Quantify User's Guide, Rational Software Corporation, 1997. URL: <http://www.rational.com/products/quantify/>
- [Salkewicz95] Salkewicz, B., "*Distributed OSPF: A Design Document*", company proprietary, Bay Networks, March 1995
- [Selic94] Selic, B., et al., "*Real-Time Object-Oriented Modeling*", John Wiley and Sons, April 1994.
- [Sidhu93] Sidhu, D., et al., "*Open Shortest Path First (OSPF) Routing Protocol Simulation*", SIGCOMM'93, Ithaca, N.Y., pp. 53-62, September 1993.
- [SNiFF+] SNiFF+ source code engineering tool, TakeFive Software, URL: <http://www.takefive.com>
- [Source Navigator] Source Navigator code analysis tool, Cygnus Solutions, URL: <http://www.cygnum.com/sn/>
- [Stallings97] Stallings, W., "*High-Speed Networks : Tcp/Ip and Atm Design Principles*", Prentice Hall College Div., 1997.

- [Tiers97] Tiers Source Code,
URL: <ftp://ftp.nexen.com/pub/papers>
- [Thomas98] Thomas, M., "*OSPF Network Design Solutions*", Cisco Press, 1998.
- [Weeks98] Weeks, G., "*IP Routing Protocols: A Technical White Paper*", company proprietary, Nortel Networks, August 1998.
- [Weiss96] Weiss, M. A., "*Algorithms, Data Structures, and Problem Solving With C++*", Addison-Wesley, 1996.
- [Wollman95] Wollman, W., Barsoum, Y., "*Overview of Open Shortest Path First, Version 2 (OSPF V2) Routing in the Tactical Environment*", MILCOM '95, Conference Record, IEEE, Vol. 3, pp. 925 -930, 1995.
- [Zegura97] Zegura, E., et al., "*A Quantitative Comparison of Graph-Based Models for Internet Topology*", IEEE/ACM Transactions on Networking, Vol. 5, No. 6, pp. 770-783, December 1997.
- [Zhu98] Zhu, S., Huang, G. M., "*A New Parallel and Distributed Shortest Path Algorithm for Hierarchically Clustered Data Networks*", IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 9, pp. 841-855, September 1998.

Appendix A

Detailed Example of OSPF Adjacency Forming Process

In OSPF, Hello packets are used to discover and maintain neighbor relationships, and also ensure that communication between neighbors is bi-directional. On multi-access networks, Hello packets are used to elect the designated router and the backup designated router. Hello packets are sent periodically over all router interfaces.

After a neighbor has been discovered, bi-directional communication ensured, and designated router elected (on broadcast or non-broadcast multiaccess networks), a decision is made regarding whether or not an adjacency should be formed with the neighbor. If an adjacency is to be formed, the first step is to synchronize the neighbors' link-state databases.

The database exchange process is used to synchronize the link-state databases between two adjacent OSPF routers when they first start communicating. At the beginning of the process, the two routers form a master/slave relationship. Then, each router describes its database by sending a sequence of Database Description packets to its neighbor. Each Database Description packet describes a set of LSAs belonging to the router's database. If the other router's database description differs, the two routers will exchange link-state requests, updates, and acknowledgements to ensure that their databases are synchronized with the most current information about the area's topology.

When the database exchange process has completed and all link-state requests have been satisfied, the databases are considered synchronized and the routers are declared to be fully adjacent.

An illustration of the OSPF database exchange process, which was adopted from [Thomas98], is presented in Figure A.1, which occurs in five steps as detailed in the numbered sequence following the figure.

Figure A.1: An example on the process of forming adjacency between two OSPF routers [Thomas98]

1. Establish bi-directional (2-way) communication. Accomplished by the discovery of neighbor routers and the election of a Designated Router (DR).
2. The two routers form a master/slave relationship and agree upon a starting sequence that will be incremented to ensure LSAs are acknowledged properly and no duplication occurs. Database Description (DD) packets begin.
3. Database Description (DD) packets continue to flow as the slave router acknowledges the master's packets.
4. Link-state requests are sent to the neighbor asking for recent advertisements that have not been discovered. At this stage, the router builds several lists to ensure all links are up-to-date and have been acknowledged properly.
5. The two routers are fully adjacent because their link-state databases are fully synchronized.

Appendix B

Optimization Solutions for the findLSA Function

From the detailed analysis of Dijkstra's algorithm in Section 5.4.4, it was found that the **findLSA** function is one of the most expensive components. This is due to the repeated and high number of calls to this function. A substantial reduction in the execution cost of Dijkstra's algorithm can be achieved by finding a way to reduce the high number of calls to the **findLSA** function.

The high number of unnecessary calls to **findLSA** were eliminated by using more efficient data structures that facilitate caching of pointers to neighboring nodes. Accordingly, pointers to neighboring nodes will be calculated once and cached for subsequent accesses. This appendix describes the design and the implementation issues of the proposed solution.

Overview of Relevant Data Structures

Figure B.1 shows the relationship between the LSDB and LSA data structures. The LSDB data structure represents a vertex in the topology graph. The data associated with a vertex includes its distance from the source router. Also, each vertex has links connecting it to adjacent vertexes. The link information (i.e., neighbor ID, and link cost) is maintained within a Link-State Advertisement (LSA) data structure. The LSDB data structure of a vertex contains a pointer to its associated LSA data structure.

Figure B.1: The relationship between the LSDB and LSA data structures

Details of the Solution

The LSA data structure was modified so that it also contains an extra space to store pointers to neighboring nodes. For each link element inside the LSA, there will be a corresponding pointer element inside this LSA, which will be initialized at some later time to point to a neighboring node. The extra space for the pointers is reserved when memory to store the incoming Link-State Advertisement is allocated. Figure B.2 shows the modifications done to the LSA data structure to support the caching of pointers. For each link entry inside the LSA data structure, there is a corresponding pointer entry inside the structure.

Figure B.2: The modifications done to the LSA data structure to facilitate caching of pointers to neighbor nodes

A Simple Example

The following is a concrete example demonstrating how the caching solution works in practice. Figure B.3 shows a simple broadcast network (i.e., an Ethernet) with three routers attached to it. Figure B.4 shows how this network will be represented in the OSPF topology graph [Comer98, Moy98b]. Each node in the graph will have its corresponding LSDB and LSA data structures. Figure B.5 shows how the various data structures will be interconnected (during runtime) through the cached pointers. Each node maintains cached pointers to its neighbors.

Figure B.3: Physical network view of a simple OSPF area

Figure B.4: The corresponding topology graph for the simple OSPF area shown in Figure B.3

Figure B.5: The interconnections between the various data structures (during runtime)

During the first run of Dijkstra's algorithm, the cached pointers to the various vertexes will be initialized through calls to the **findLSA** function. However, subsequent runs of Dijkstra's algorithm will not need to call the **findLSA** function, since from the LSA of each vertex (node), neighbor nodes are immediately found by following the corresponding cached pointer to each link. Thus, this solution this will eliminate the repeated calls to the **findLSA** function.

Some issues with the proposed solution

The previous solution was based on the assumption that the link-state database remains intact and the neighbors of a node remain the same and don't change over time. This is not a realistic assumption; link-state databases change over time. If there are changes to the link-state database, such as the deletion of an LSA by the aging process, then the previously cached pointers to the deleted nodes will be invalid. The question is how to keep the cached pointers up-to-date after changes to the link-state database?

The proposed solution is: whenever there is an event that changes the link-state database, then from the specific node which has changed, find out its neighbors, and update their cached pointers to the changed node.

The following is a list of the different cases that will result in changes to the link-state database, that shows how to update/reset the cached pointers if required:

- **The aging of an LSA:** There is an age limit on each LSA, which is periodically incremented, and if an LSA exceeds its maximum allowable age, it will be deleted from the link-state database. The maximum allowable age for an LSA is 1 hour [Moy98b]. Since the aged LSA will be deleted from the link-state database, any previously cached pointers to this aged LSA will be invalid. The solution is to find out the neighbors of this aged node (LSA), and reset their cached pointers to the aged node. The neighbors of an aged node can be easily found by using the cached pointers that are maintained inside the aged LSA itself.
- **The reception of a Link-State Advertisement as a replacement to previously installed version in the link-state database:** Even though the installed LSA in the link-state database will be replaced by the newer LSA, any previously cached pointers to the associated node will be still valid. The pointers are pointing to a LSDB data structure, not the LSA data structure associated with this LSDB. Since the LSA is the one that will be replaced, and not the LSDB structure, then there are no cached pointers to update or reset.
- **The reception of a totally new Link-State Advertisement:** Similar to the previous case, there are no cached pointers to update or reset in this case too. Since this is a totally new LSA and there is no any previous version of this LSA installed in the link-state database, there will not be any previously cached pointers to it.

Appendix C

Illustrative Scenarios on the Effects of Topological Changes within an OSPF Area

This Appendix discusses four scenarios that illustrate some typical topological changes in the OSPF area, which affect the link-state database and lead to recomputing the routing table at each router inside the area.

Whenever a router detects a change in its formed adjacencies, it creates a new LSA and floods it into the entire area [Moy98b]. To better understand this process, let us look at the following four simple scenarios occurring in the OSPF area depicted in Figure 2.1. These scenarios are representative of the common topological changes that happen in an OSPF area. It is assumed that scenario A2 occurs after scenario A1, and that scenario B2 occurs after scenario B1.

Scenario A1: The link between R4 and R6 is down

In our first scenario, the link between routers R4 and R6 goes down. Both routers will detect that the link is down, generate a new LSA and flood it throughout the area. After the other routers inside the area receive the two LSAs, they will replace the last LSAs, generated by R4 and R6, in their link-state databases with the newer LSAs. Figure C.1 shows the LSAs generated by R4 and R6, and the new topology graph resulting from the topological change in the area.

Figure C.1: OSPF LSAs exchanged after the link between R4 and R6 fails, and the changes in the area topology in scenario A1

Now let us discuss the specific case when router R1 receives the two new LSAs generated by R4 and R6. The *Topology Change Detector* will be invoked. This component will analyze the two LSAs and determine the specific topological change that has triggered their generation. Then it will populate the topology change list with change events. Figure C.2 shows the topology change list created by the *Topology Change Detector* as a result of this scenario. When the *ISPF Algorithm* runs, it will examine the list of change events and will update the shortest path tree and the Candidate List as required. Figure C.3 shows the nodes added to the Candidate List by the *ISPF Algorithm*. Finally, the *SPF Algorithm* will run and, using the shortest path tree and the Candidate List, it will update the routing table. Figure C.4 shows the new SPF tree.

Figure C.2: The topology change list for scenario A1

Figure C.3: The Candidate List for scenario A1

Figure C.4: The SPF tree for scenario A1

Scenario A2: The link between R4 and R6 is now up

Now, in a second scenario, the link between routers R4 and R6 is up again. Both routers will detect that the link is up, generate a new LSA and flood it throughout the area. The LSAs generated by R4 and R6 are identical to those shown in Figure 2.3. After the other routers inside the area receive these two LSAs, they will replace the last LSAs, generated by R4 and R6, in their link-state databases with the newer LSAs. The resulting topology graph is the same as the one shown in Figure 2.2.

Figure C.5 shows the topology change list created by the *Topology Change Detector* and Figure C.6 shows the nodes added to the Candidate List by the *ISPF Algorithm* as

a result of router R1 receiving the new LSAs. The new SPF tree is identical to the one shown in Figure 2.4.

Figure C.5: The topology change list for scenario A2

Figure C.6: The Candidate List for scenario A2

Scenario B1: Router R6 is down

In the third scenario, router R6 goes down. Each neighbor of R6 (R4, R5 and R7) will detect that R6 is down, generate a new LSA and flood it throughout the area. After the other routers inside the area receive the three LSAs, they will replace the last LSAs, generated by R4, R5 and R7 in their link-state databases with the newer LSAs. Figure C.7 shows the LSAs generated by R4, R5 and R7, the new topology graph resulting from the changes in the area.

Figure C.7: OSPF LSAs exchanged after R6 fails, and the changes in the area topology in scenario B1

Figure C.8 shows the topology change list created by the *Topology Change Detector* and Figure C.9 shows the nodes added to the Candidate List by the *ISPF Algorithm* as a result of router R1 receiving the new LSAs. Figure C.10 shows the new SPF tree.

Figure C.8: The topology change list for scenario B1 **Figure C.9: The Candidate List for scenario B1**

Figure C.10: The SPF tree for scenario B1

Notice that the two LSAs generated by R4 in Scenarios A1 and B1 are identical. The reason is that the two events are closely related. The failure of the link between R6 and R4, and the failure of R6, will have the same effect from R4's point-of-view. In both cases R6 is unreachable directly from R4.

A final note, the LSA of router R6 will eventually be deleted from the link-state databases of all routers inside the OSPF area when it reaches its maximum allowable age, which is one hour [Moy98b].

Scenario B2: Router R6 is now up

In the final scenario, router R6 comes up again. Each neighbor of R6 (R4, R5 and R7) will detect that R6 is up and generate a new LSA to reflect this fact. R6 will synchronize its link-state database with its neighbors and will generate a new LSA for itself. All four new LSAs will be flooded throughout the entire area. The LSAs generated by R4, R5, R7 and R6 are identical to those in Figure 2.3. After the other routers inside the area receive these four LSAs, they will replace the last LSAs, generated by R4, R5, R7 and R6 (if it didn't expire yet), in their link-state databases with the new LSAs. The resulting topology graph is the same as the one shown in Figure 2.2.

Figure C.11 shows the topology change list created by the *Topology Change Detector* and Figure C.12 shows the nodes added to the Candidate List by the *ISPF Algorithm* as a result of router R1 receiving the new LSAs. The new SPF tree is identical to the one shown in Figure 2.4.

Figure C.11: The topology change list for scenario B2

Figure C.12: The Candidate List for scenario B2